

**Department of Computer Engineering,
City College of New York**

CSC 342 – Computer Organization [CSC 342_Spring_2023]

DATE: 5/7/2023

Instructor: Professor, Izidor Gertner

TA: Professor, Donald Lushi

Student: Paniz Peiravani

**Take Home Final Test:
Optimization of dot product computation
of two vectors using vector instructions**

On Windows intel x64 processor, and Linux Ubuntu

Contents

| | |
|---|----|
| Objective | 3 |
| Task 1 - CPU ID | 3 |
| Windows – Intel x64 | 3 |
| Linux - Ubuntu..... | 5 |
| Task 2 - Compute dot product Using C++ (Non-Optimized) | 6 |
| Windows – Intel x64 | 6 |
| Run the Code..... | 10 |
| Graph | 16 |
| Linux - Ubuntu..... | 17 |
| Run the Code..... | 19 |
| Graph | 24 |
| Task 3 - Compute dot product Using C++ (Non-Optimized and Optimized)..... | 25 |
| Windows – Intel x64 | 25 |
| Non-Optimized Assembly Code | 25 |
| Compiler Optimization..... | 29 |
| Manually Optimization..... | 38 |
| Graph | 41 |
| Linux – Ubuntu..... | 42 |
| Non-Optimized Assembly Code | 42 |
| Compiler Optimization..... | 46 |
| Manually Optimization..... | 51 |
| Graph | 55 |
| Task 4 – DPPS | 56 |
| Windows – Intel x64 | 56 |
| Run the Code..... | 56 |
| Graph | 57 |
| Linux – Ubuntu..... | 58 |
| Run the Code..... | 58 |
| Graph | 59 |
| Task 5 – All Plots in on Graph..... | 60 |
| Windows – Intel x64 | 60 |
| Linux – Ubuntu..... | 62 |
| Conclusion..... | 63 |

Objective

The purpose of this final take-home assignment is to learn to optimize compiler-generated code for computing dot products using vector instructions. First, I needed to determine my processor's CPU ID. The CPU provides the instructions and processing power necessary for the computer to function. Generally, the higher the clock speed per core, the better the CPU. We will then generate C++ codes for the dot product, disabling and enabling automatic parallelization and automatic vectorization in separate instances. We will also inspect the compiler-generated assembly code to see if the compiler has vectorized the code for very large vector sizes, and then optimize it accordingly. Finally, based on the optimized code, we will generate assembly code and manually optimize our assembly code and we will use DPPS.

For each step, we will measure execution time and plot time versus vector size. Moreover, all the tasks will be generated one time on Windows, Intel(R) Core (TM) i7, using Visual Studio, and one time on Linux, Ubuntu using gcc.

Task 1 - CPU ID

Windows – Intel x64

To find my CPU ID, I can use a command prompt like *Figure 1*. However, using the command prompt does not provide detailed information about specific vector instruction sets. For finding vector instruction sets we can use CPU Z.

The below Figure shows the CPU ID using the command prompt on Windows. For finding the CPU ID from the command prompt, we can run the commands line below.

```
wmic // Windows Management Instrumentation Command-line which
      allows to access management information.
cpu get // show us information about the CPU (processor) in the
        system.
cpu get processorId // show us the Processor ID of the CPU.

((c) Microsoft Corporation. All rights reserved.

C:\Users\paniw>wmic
wmic:root\cli>cpu get
AddressWidth Architecture AssetTag Availability Caption Characteristics64
9 To Be Filled By O.E.M. 3 Intel64 Family 6 Model 126 Stepping 5 252

wmic:root\cli>cpu get processorId
ProcessorId
BFEFBFFF00E706E5
```

Figure 1 - CPU ID using command prompt

For finding detailed information about specific vector instruction sets that my CPU support, we can use CPU Z.

From CPU Z, we can find our processor name, vector sets that are available, core speed, etc.

Based on the vectors instruction we have:

MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, and SSE4.2 which are for streaming SIMD extensions.

EM64T or Intel 64 or AMD64 is for Extended Memory 64 Technology.

VT-x or virtualization technology is a hardware virtualization technology developed by Intel.

AVX, and AVX2 are for advanced vector extensions.

AVX-512 is for advanced vector extensions 512-bit.

FMA3 is for fused multiply-add 3.

SHA or secure hash algorithm is a family of cryptographic hash functions used for data integrity and authentication.

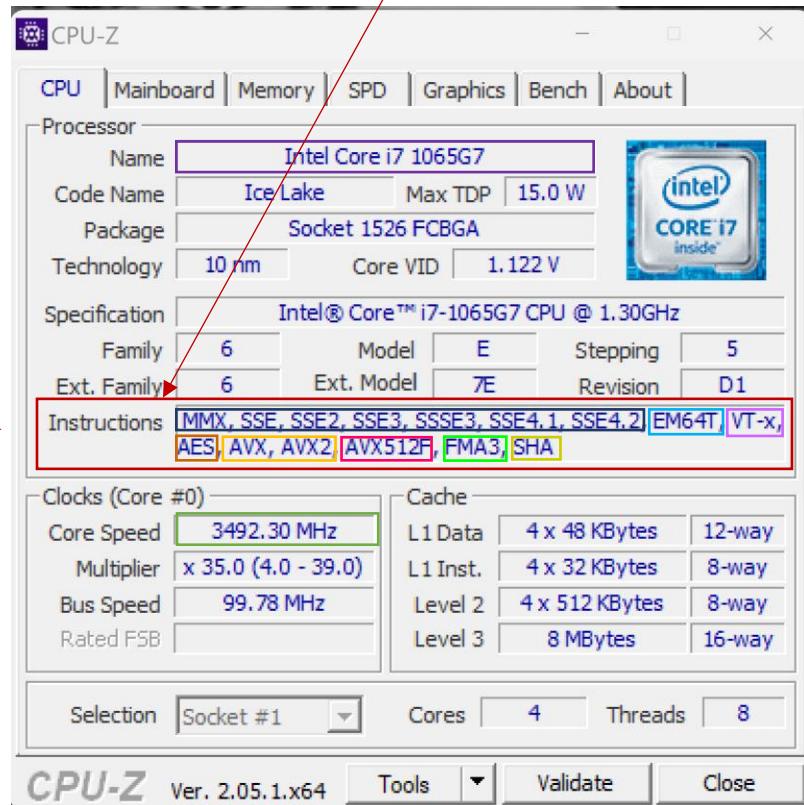


Figure 2 - CPU ID using CPU Z on Windows

Linux - Ubuntu

To find the CPU ID on Linux, we can run the command line on the Linux terminal.

To find the CPU Model we can run below command line:

```
lscpu | grep "Model name"          // CPU architecture information | global regular expression print
```

```
paniz@paniz-VirtualBox:~/Desktop$ lscpu | grep "Model name"
Model name:           Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
paniz@paniz-VirtualBox:~/Desktop$ █
```

Figure 3 - CPU ID on Linux

To find the CPU ID with specifications for vector processing capabilities, we can run the below command:

```
Lscpu          // CPU architecture information
```

In *Figure 4*, from Flags we can find the vector specification.

```
paniz@paniz-VirtualBox:~/Desktop$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:         39 bits physical, 48 bits virtual
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:  0-3
Vendor ID:             GenuineIntel
Model name:            Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
CPU family:            6
Model:                 126
Thread(s) per core:   1
Core(s) per socket:   4
Socket(s):            1
Stepping:              5
BogoMIPS:              2995.20
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mc
a cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 cx16 pci d sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm 3dnowprefetch invpcid_single fsgsbase bmi1 avx2 bmi2 invpcid rdseed clflushopt md_clear flush_l1d arch_capabilities

Virtualization features:
Hypervisor vendor:    KVM
Virtualization type:  full
Caches (sum of all):
L1d:                  192 KiB (4 instances)
L1i:                  128 KiB (4 instances)
L2:                   2 MiB (4 instances)
L3:                   32 MiB (4 instances)
NUMA:
NUMA node(s):         1
```

Figure 4 - CPU ID for vector specification on Linux

```

Normal Node(s): 1
NUMA node0 CPU(s): 0-3
Vulnerabilities:
  Icelake multithit: KVM: Mitigation: VMX unsupported
  L1tf: Not affected
  Mds: Not affected
  Meltdown: Not affected
  Mmio stale data: Vulnerable: Clear CPU buffers attempted, no microcode;
                     SMT Host state unknown
  Retbleed: Vulnerable
  Spec store bypass: Vulnerable
  Spectre v1: Mitigation: usercopy/swapgs barriers and __user pointer
               sanitization
  Spectre v2: Mitigation: Retpolines, STIBP disabled, RSB filling, PB
  Srbds: RSB-eIBRS Not affected
  Tsx async abort: Unknown: Dependent on hypervisor status
  Not affected

paniz@paniz-VirtualBox:~/Desktop$
```

Figure 5 – Rest of the CPU ID for vector specification on Linux

As you see we have similar vector processing capabilities since Ubuntu is installed on a virtual machine on my Windows device with the same hardware, including the CPU. Hence, CPU remains the same for both the host operating system, Windows, and the guest operating system, Ubuntu.

Task 2 - Compute dot product Using C++ (Non-Optimized)

Windows – Intel x64

Based on the requirement, first we need to disable automatic parallelization and automatic vectorization. To disable automatic parallelization and automatic vectorization, we need to do the project properties. In order to get to the project properties, we need to right-click on our project name.

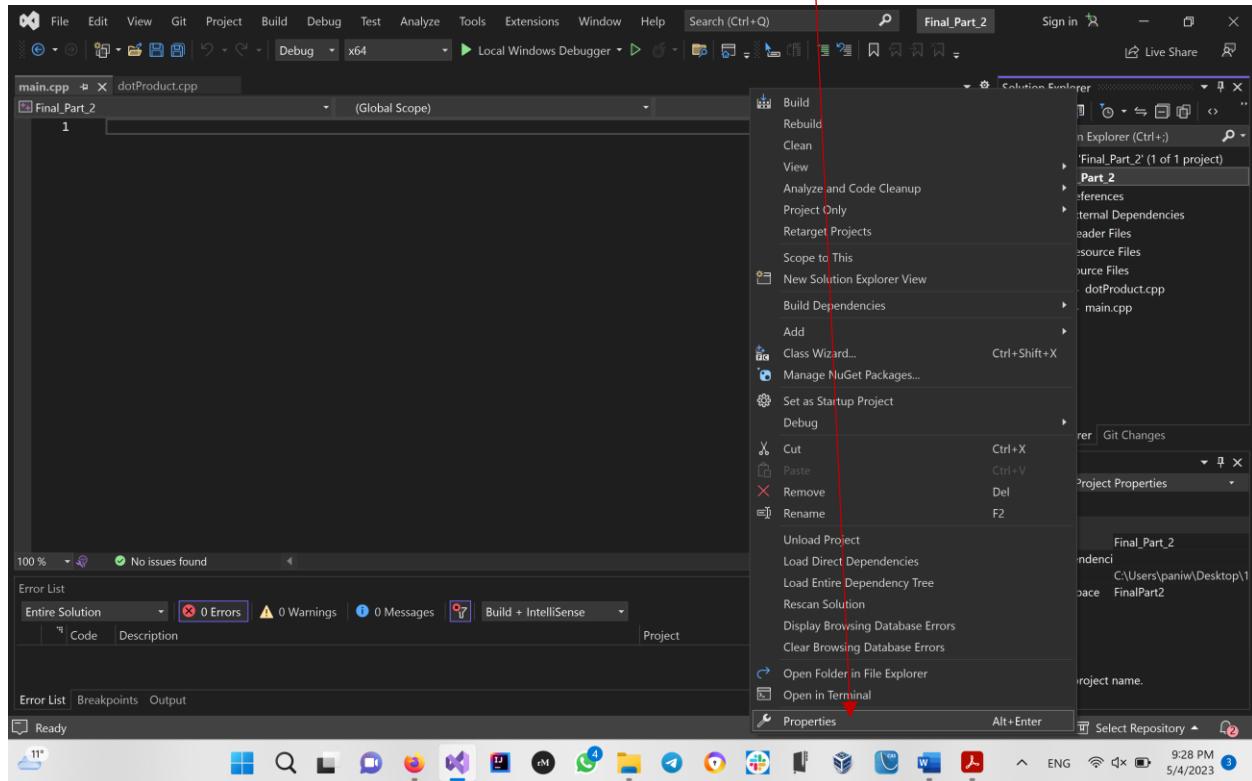


Figure 6 - Project Properties on VS

Once we are in project properties, we can use the steps below to disable Parallel Code Generation and Enhanced Instruction Set.

Project Property, C/C++, Code Generation, and enable Parallel Code Generation by using *No (/Qpar-)* and enable Enhanced Instruction Set by using *No Enhanced Instructions (/arch:IA32)* and then press apply.

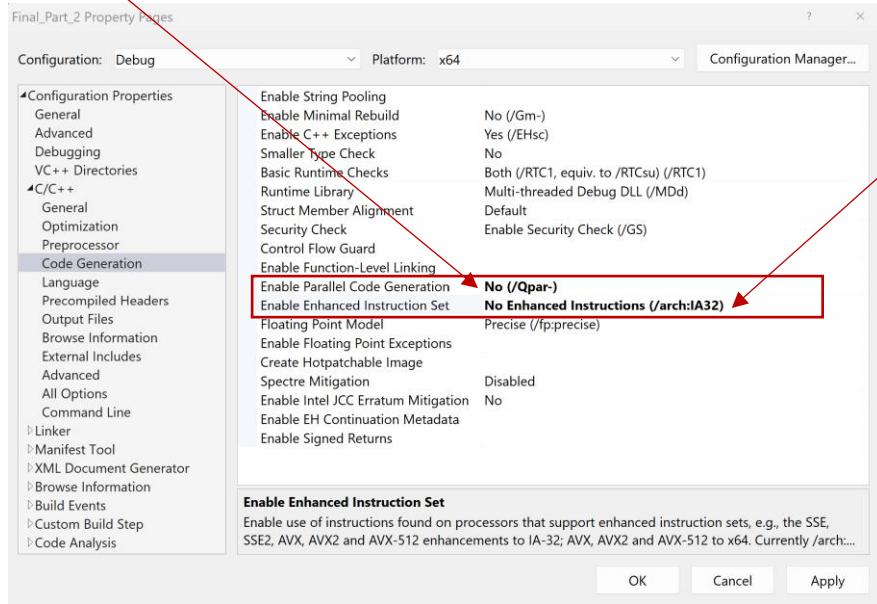


Figure 7 - Enable Parallel Code Generation and Enhanced Instruction Set on VS

We need to also make sure that optimization is not on. We can use the steps below to make sure optimization is unable.

Project Property, C/C++, Optimization, Disable (/Od)

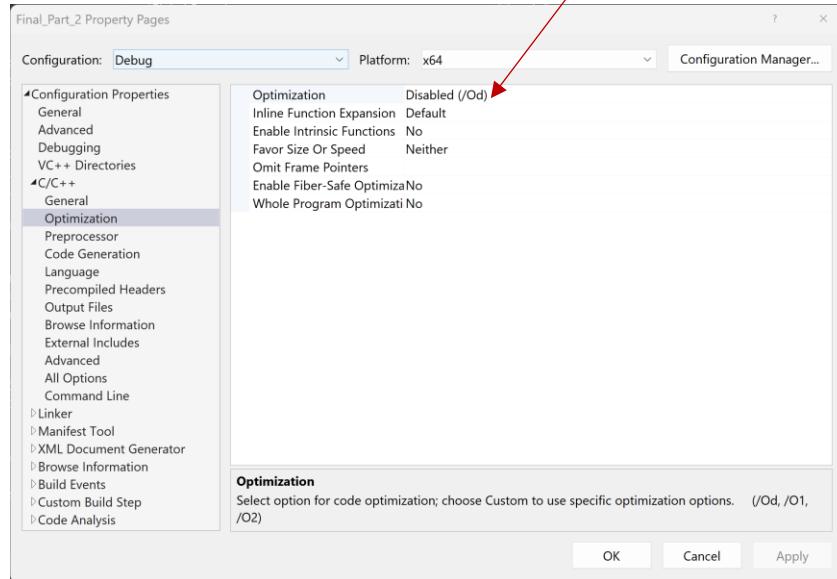


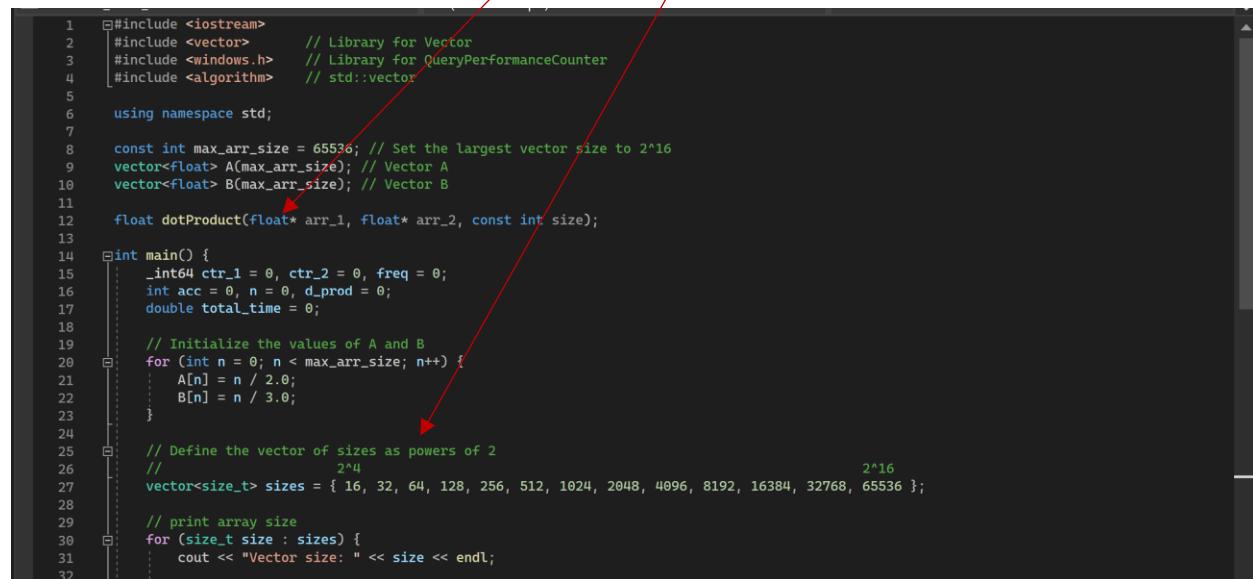
Figure 8 – Disable Optimization on VS

We will have two code files; for instance, we will have a source file - dotProduct.cpp, and a main file - main.cpp. The dotProduct.cpp contains the definition of the dotProduct function which means it is the implementation calculation. The main.cpp contains the vector size and defines a vector of powers of 2 size, and printouts value.

Below on *Figure 9*, you can see the screenshot from my Visual Studio for our main.cpp function. Since we want to use vector, we need to include `<vector>` and `<algorithm>` library, and for QueryPerformanceCounter we need to include `<windows.h>` library. Next, we will have our array which is set to the largest vector size which is 2^{16} for this assignment, and initial our arrays A and B. On line 12, we have a definition of dotProduct, and starting line 14 we will start the main function.

In our main function we declare counter variables, dot product result, and total time and initial all of them to 0. Next, we are initiating the value of our two arrays which are A and B.

We will also define our array size which is from 2^4 to 2^{16} based on the requirements and print the array size.



```

1  #include <iostream>
2  #include <vector>           // Library for Vector
3  #include <windows.h>        // Library for QueryPerformanceCounter
4  #include <algorithm>         // std::vector
5
6  using namespace std;
7
8  const int max_arr_size = 65536; // Set the largest vector size to 2^16
9  vector<float> A(max_arr_size); // Vector A
10 vector<float> B(max_arr_size); // Vector B
11
12 float dotProduct(float* arr_1, float* arr_2, const int size);
13
14 int main() {
15     _int64 ctr_1 = 0, ctr_2 = 0, freq = 0;
16     int acc = 0, n = 0, d_prod = 0;
17     double total_time = 0;
18
19     // Initialize the values of A and B
20     for (int n = 0; n < max_arr_size; n++) {
21         A[n] = n / 2.0;
22         B[n] = n / 3.0;
23     }
24
25     // Define the vector of sizes as powers of 2
26     //          2^4                                2^16
27     vector<size_t> sizes = { 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 };
28
29     // print array size
30     for (size_t size : sizes) {
31         cout << "Vector size: " << size << endl;
32     }

```

Figure 9 - C++ on VS

We will run dotProduct 10 times and for each size. We will use Query to calculate the min resolution, execution time, average time, and total time in seconds and print the values. We also call the dotProduct function that we created in our dotProduct.cpp.

```

32 // Run the dotProduct function 10 times for each size and calculate time
33 for (int i = 0; i < 10; i++) {
34     // Start timing the code
35     if (QueryPerformanceCounter((LARGE_INTEGER*)&ctr_1) != 0) {
36         // Call the dotProduct function
37         d_prod = dotProduct(&A[0], &B[0], size);
38
39         // Finish timing the code.
40         QueryPerformanceCounter((LARGE_INTEGER*)&ctr_2);
41
42         // Get the performance counter frequency
43         QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
44
45         // min_resolution
46         cout << "QueryPerformance min resolution: 1/" << freq << " seconds" << endl;
47
48         // Calculate the execution time in seconds
49         cout << "Run number: " << i + 1 << endl << "Running time: " << (ctr_2 - ctr_1) * 1.0 / freq << " seconds" << endl;
50         total_time += (ctr_2 - ctr_1) * 1.0 / freq;
51
52     }
53
54     else {
55         DWORD dwError = GetLastError();
56         cout << "Error value: " << dwError << endl;
57     }
58 }
59
60 // Print the average time for the current size
61 cout << "Average time: " << (total_time / 10) << endl;
62 cout << "TotalTime: " << total_time << endl << endl;
63 cout << "-----" << endl;
64
65 system("pause");
66 return 0;
67
68
69

```

Figure 10 - C++ on VS

Run the Code

After we run the code, we will get the below results. *Figure 11* states the result for vector size 2^4 or 16. We can see that the dotProduct ran 10 times and for each time we can see the running time. To make it easier to follow the figures, I added boxes to *Figure 11* to improve clarity and help convey the idea behind each figure. If you look at the Figures, you can see the min resolution stays the same. The reason that we get the same result for the min resolution every time is that the performance counter frequency `freq` is a fixed value for my system. For instance, it is determined by the hardware.

```
Vector size: 16
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 1.1e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 1e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 1.2e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 1e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 8e-07 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 6e-07 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 6e-07 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 6e-07 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 6e-07 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 7e-07 seconds
Average time: 8.2e-07
TotalTime: 8.2e-06
```

Figure 11 – dot Product for 2^4 without /Qpar and /arch

```
Vector size: 32
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 1.5e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 1.3e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 1.2e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 1.3e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 1.2e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 9e-07 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 1.2e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 9e-07 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 1e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 1.7e-06 seconds
Average time: 2.04e-06
TotalTime: 2.04e-05
```

Figure 12 - dot Product for 2^5 without /Qpar and /arch

```

Vector size: 64
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 1.3e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 2.5e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 2.3e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 1.7e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 1.9e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 1.9e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 1.7e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 1.4e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 1.5e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 1.8e-06 seconds
Average time: 3.84e-06
TotalTime: 3.84e-05
-----
```

Figure 13 - dot Product for 2^6 without /Qpar and /arch

```

Vector size: 128
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 1.19e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 2.2e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 1.8e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 1.8e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 1.6e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 1.5e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 1.5e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 1.6e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 1.6e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 3.2e-06 seconds
Average time: 6.71e-06
TotalTime: 6.71e-05
-----
```

Figure 14 - dot Product for 2^7 without /Qpar and /arch

```

Vector size: 256
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 5.1e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 5e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 3.4e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 4.6e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 1.48e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 4.4e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 5.1e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 5.1e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 8.8e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 4.4e-06 seconds
Average time: 1.278e-05
TotalTime: 0.0001278
-----
```

Figure 15 - dot Product for 2^8 without /Qpar and /arch

```

Vector size: 512
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 1.45e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 1.03e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 7.3e-06 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 1.43e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 1.27e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 4.22e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 1.31e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 1.34e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 1.31e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 0.000136 seconds
Average time: 4.047e-05
Totaltime: 0.0004047
-----
```

Figure 16 - dot Product for 2^9 without /Qpar and /arch

```

Vector size: 1024
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 2.38e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 3.19e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 3.51e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 2.66e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 2.32e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 2.56e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 1.45e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 3e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 2.4e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 9.33e-05 seconds
Average time: 7.327e-05
TotalTime: 0.0007327
-----
```

Figure 17 - dot Product for 2^{10} without /Qpar and /arch

```

Vector size: 2048
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 5.14e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 6.52e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 5.66e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 4.61e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 3.65e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 7.24e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 5.69e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 3.93e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 4.36e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 4.79e-05 seconds
Average time: 0.00012486
Totaltime: 0.0012486
-----
```

Figure 18 - dot Product for 2^{11} without /Qpar and /arch

```

Vector size: 4096
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 0.0001238 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 0.0001109 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 0.0001218 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 0.0001067 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 0.000167 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 7.35e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 8.51e-05 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 0.0001043 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 0.000116 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 0.0001782 seconds
Average time: 0.00024359
TotalTime: 0.0024359
-----
```

Figure 19 - dot Product for 2^{12} without /Qpar and /arch

```

Vector size: 8192
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 0.0002789 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 0.0002496 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 0.0002122 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 0.000207 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 0.0002341 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 0.0002034 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 0.0003421 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 0.0002963 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 0.0002161 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 0.000195 seconds
Average time: 0.00048706
TotalTime: 0.0048706
-----
```

Figure 20 - dot Product for 2^{13} without /Qpar and /arch

```

Vector size: 16384
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 0.0004109 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 0.0004375 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 0.0006299 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 0.0003863 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 0.0004206 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 0.0003839 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 0.0003555 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 0.0003972 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 0.0003364 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 0.0003756 seconds
Average time: 0.00090044
TotalTime: 0.0090044
-----
```

Figure 21 - dot Product for 2^{14} without /Qpar and /arch

```

Vector size: 32768
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 0.0005503 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 0.000755 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 0.0004873 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 0.000865 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 0.0005984 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 0.0005501 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 0.0005117 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 0.0005487 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 0.0003456 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 0.0002878 seconds
Average time: 0.00145043
TotalTime: 0.0145043
-----
```

Figure 22 - dot Product for 2^{15} without /Qpar and /arch

```

Vector size: 65536
QueryPerformance min resolution: 1/10000000 seconds
Run number: 1
Running time: 0.0005912 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 2
Running time: 0.0008863 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 3
Running time: 0.0008584 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 4
Running time: 0.0006314 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 5
Running time: 0.0006103 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 6
Running time: 0.0006761 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 7
Running time: 0.0008685 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 8
Running time: 0.0008371 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 9
Running time: 0.0007439 seconds
QueryPerformance min resolution: 1/10000000 seconds
Run number: 10
Running time: 0.000592 seconds
Average time: 0.00217995
TotalTime: 0.0217995
-----
```

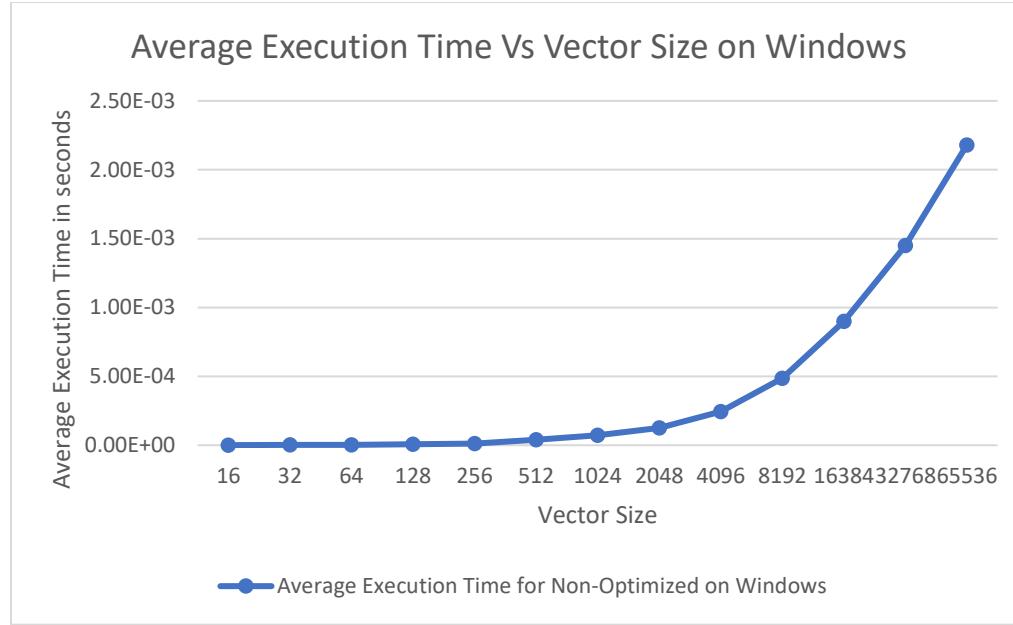
Figure 23 - dot Product for 2^{16} without /Qpar and /arch

Graph

Now that we have execution time, we will graph Average Execution time Vs Vector Size for unoptimized code. We ran the code 10 times for each vector size to be able to find the average execution time that is more accurate.

Table 1 - Task 2 - Windows

| Vector Size | Average Execution Time in seconds for Non-Optimized on Windows |
|-------------|--|
| 16 | 8.20E-07 |
| 32 | 2.04E-06 |
| 64 | 3.84E-06 |
| 128 | 6.71E-06 |
| 256 | 1.28E-05 |
| 512 | 4.05E-05 |
| 1024 | 7.33E-05 |
| 2048 | 1.25E-04 |
| 4096 | 2.44E-04 |
| 8192 | 4.87E-04 |
| 16384 | 9.00E-04 |
| 32768 | 1.45E-03 |
| 65536 | 2.18E-03 |



Graph 1 – Average Execution Time Vs Vector Size for Non-Optimized on Windows

Linux - Ubuntu

We are using the same C++ codes for the function of dot product. However, some modifications are needed for main.cpp. For instance, the C++ code on Visual Studio uses QueryPerformanceCounter function which is Windows API for measuring time; hence, we need to change it to something that be equivalent of Query but for Linux. We can use the clock_gettime function which is equivalent of QueryPerformanceCounter for Linux.

As you can see in *Figure 24*, first we removed the `#include <windows.h>` since now we are using Linux and then used `#include <time.h>` which is a time library in Linux that we need to for `clock_gettime`. The rest of the code stays the same until line 43, which we start to find the execution time.

```

1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 #include <time.h> // Line 4
5
6 using namespace std;
7
8 const int max_arr_size = 65536; // Set the largest vector size to 2^16
9 vector<float> A(max_arr_size); // Vector A
10 vector<float> B(max_arr_size); // Vector B
11
12 // Function to calculate the dot product of two vectors
13 float dotProduct(const std::vector<float>& arr_1, const std::vector<float>& arr_2, const int size);
14
15 int main() {
16     timespec start_time, end_time;
17     double elapsed_time, total_time;
18     int d_prod = 0;
19
20     // Initialize the values of A and B
21     for (int n = 0; n < max_arr_size; n++) {
22         A[n] = n / 2.0;
23         B[n] = n / 3.0;
24     }
25
26     // Define the vector of sizes as powers of 2
27     vector<size_t> sizes = { 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536 };
28
29     // Iterate over each vector size
30     for (size_t size : sizes) {
31         cout << "Vector size: " << size << endl;
32         total_time = 0;
33     }
34 }
```

Figure 24 - C++ on Ubuntu gcc

Starting line 34, we are starting to find the execution time by using `clock_gettime`. We also need to divide the execution time by `1e9` to get the time in seconds since the `tv_nsec` field stores the time in nanosecond.

```

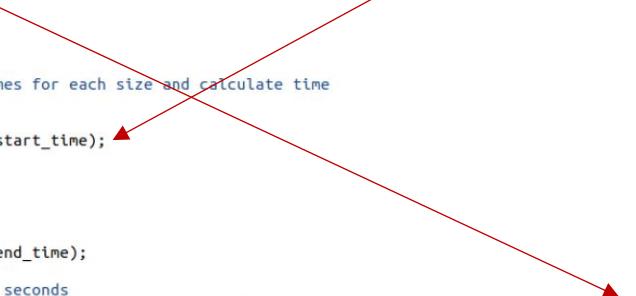
33     -
34     // Run the dotProduct function 10 times for each size and calculate time
35     for (int i = 0; i < 10; i++) {
36         // Start timing the code
37         clock_gettime(CLOCK_MONOTONIC, &start_time); 
38
39         // Call the dotProduct function
40         d_prod = dotProduct(A, B, size);
41
42         // Finish timing the code
43         clock_gettime(CLOCK_MONOTONIC, &end_time);
44
45         // Calculate the elapsed time in seconds
46         elapsed_time = (end_time.tv_sec - start_time.tv_sec) + (end_time.tv_nsec - start_time.tv_nsec) / 1e9;
47         cout << "Run number: " << i + 1 << endl << "Running time: " << elapsed_time << " seconds" << endl;
48         total_time += elapsed_time;
49     }
50
51     // Print the average time for the current size
52     cout << "Average time: " << (total_time / 10) << endl;
53     cout << "TotalTime: " << total_time << endl << endl;
54     cout << "-----" << endl;
55 }
56
57 return 0;
58 }
```

Figure 25 - C++ on Ubuntu gcc

The `dotProduct.cpp` stays the same as before since it is just definition of `dotProduct`. We will call the `dotProduct.cpp` in `main.cpp` on line 40.

```

1 #include <vector>
2
3 // Definition of the dotProduct function
4 float dotProduct(const std::vector<float>& arr_1, const std::vector<float>& arr_2, const int size) {
5     float result = 0;
6
7     // Iterate through both vectors and calculate the dot product
8     for (int i = 0; i < size; i++) {
9         result += arr_1[i] * arr_2[i];
10    }
11    return result;
12 }
```

Figure 26 - C++ on Ubuntu gcc

Now that we have our code, we need to disable the automatic parallelization and automatic vectorization. For disabling automatic parallelization and automatic vectorization we can use the below command line on Ubuntu gcc.

```

// The -fno-tree-vectorize flag disables automatic vectorization.

// The -fno-tree-loop-distribute-patterns flag disables automatic loop
parallelization.

// We need to link out dotProduct and main as well

// The -O0 will make sure that the code stays unoptimized

g++ -O0 -fno-tree-loop-distribute-patterns -fno-tree-vectorize
Peiravani_Paniz_main.cpp Peiravani_Paniz_dotProduct.cpp -o main
```

```

paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ g++ -O0 -fno-tree-loop-distribute-patterns -fno-tree-vectorize
Peiravani_Paniz_main.cpp Peiravani_Paniz_dotProduct.cpp -o main
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ ./main

```

Figure 27 - Compile the C++ code on Linux

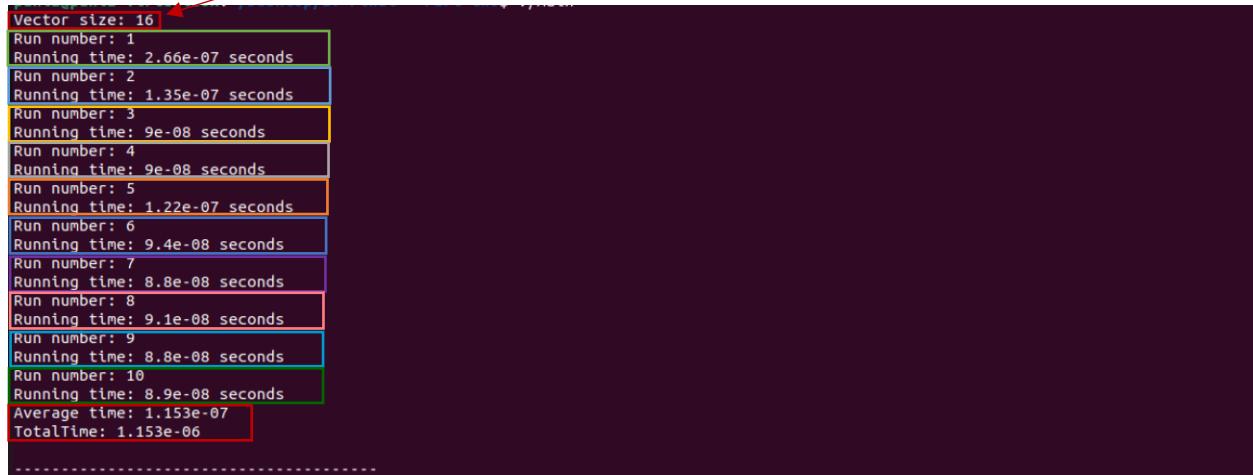
Linux gcc has command line flags that we can use for optimizations.

Table 2 - Optimization Flags on Linux gcc

| Optimization Flags | Optimizations Level | Optimizations |
|--------------------|---------------------|----------------|
| -O0 | O (Default) | None |
| -O1 | 1 | Basic |
| -O2 | 2 | Recommended |
| -O3 | 3 (Highest) | Not Guaranteed |

Run the Code

After we run the code by using `./main` which is our `a.out` file, we will get the below results. *Figure 28* states the result for vector size 2^4 or 16. We can see that the `dotProduct` ran 10 times and for each time we can see the running time. To make it easier to follow the figures, I added boxes to *Figure 28* to improve clarity and help convey the idea behind each figure.



```

Vector size: 16
Run number: 1
Running time: 2.66e-07 seconds
Run number: 2
Running time: 1.35e-07 seconds
Run number: 3
Running time: 9e-08 seconds
Run number: 4
Running time: 9e-08 seconds
Run number: 5
Running time: 1.22e-07 seconds
Run number: 6
Running time: 9.4e-08 seconds
Run number: 7
Running time: 8.8e-08 seconds
Run number: 8
Running time: 9.1e-08 seconds
Run number: 9
Running time: 8.8e-08 seconds
Run number: 10
Running time: 8.9e-08 seconds
Average time: 1.153e-07
TotalTime: 1.153e-06
-----
```

Figure 28 - dot Product for 2^4 on Linux



```

Vector size: 32
Run number: 1
Running time: 2.26e-07 seconds
Run number: 2
Running time: 2.99e-07 seconds
Run number: 3
Running time: 2.57e-07 seconds
Run number: 4
Running time: 2.37e-07 seconds
Run number: 5
Running time: 2.67e-07 seconds
Run number: 6
Running time: 2.24e-07 seconds
Run number: 7
Running time: 2.46e-07 seconds
Run number: 8
Running time: 2.11e-07 seconds
Run number: 9
Running time: 2.17e-07 seconds
Run number: 10
Running time: 2.2e-07 seconds
Average time: 2.404e-07
TotalTime: 2.404e-06
-----
```

Figure 29 - dot Product for 2^5 on Linux

```

Vector size: 64
Run number: 1
Running time: 4.04e-07 seconds
Run number: 2
Running time: 4.3e-07 seconds
Run number: 3
Running time: 4.35e-07 seconds
Run number: 4
Running time: 4.01e-07 seconds
Run number: 5
Running time: 4.04e-07 seconds
Run number: 6
Running time: 4.39e-07 seconds
Run number: 7
Running time: 4e-07 seconds
Run number: 8
Running time: 5.03e-07 seconds
Run number: 9
Running time: 4.7e-07 seconds
Run number: 10
Running time: 4.38e-07 seconds
Average time: 4.324e-07
TotalTime: 4.324e-06
-----
```

Figure 30 - dot Product for 2^6 on Linux

```

Vector size: 128
Run number: 1
Running time: 7.8e-07 seconds
Run number: 2
Running time: 8.1e-07 seconds
Run number: 3
Running time: 8.53e-07 seconds
Run number: 4
Running time: 8.24e-07 seconds
Run number: 5
Running time: 8.24e-07 seconds
Run number: 6
Running time: 8.14e-07 seconds
Run number: 7
Running time: 8.35e-07 seconds
Run number: 8
Running time: 8.36e-07 seconds
Run number: 9
Running time: 8.9e-07 seconds
Run number: 10
Running time: 7.92e-07 seconds
Average time: 8.258e-07
TotalTime: 8.258e-06
-----
```

Figure 31 - dot Product for 2^7 on Linux

```

Vector size: 256
Run number: 1
Running time: 1.644e-06 seconds
Run number: 2
Running time: 1.645e-06 seconds
Run number: 3
Running time: 1.628e-06 seconds
Run number: 4
Running time: 1.487e-06 seconds
Run number: 5
Running time: 1.505e-06 seconds
Run number: 6
Running time: 1.663e-06 seconds
Run number: 7
Running time: 1.157e-06 seconds
Run number: 8
Running time: 1.014e-06 seconds
Run number: 9
Running time: 1.984e-06 seconds
Run number: 10
Running time: 1.51e-06 seconds
Average time: 1.5237e-06
TotalTime: 1.5237e-05
-----
```

Figure 32 - dot Product for 2^8 on Linux

```
Vector size: 512
Run number: 1
Running time: 4.714e-06 seconds
Run number: 2
Running time: 3.113e-06 seconds
Run number: 3
Running time: 2.767e-06 seconds
Run number: 4
Running time: 1.958e-06 seconds
Run number: 5
Running time: 1.944e-06 seconds
Run number: 6
Running time: 1.943e-06 seconds
Run number: 7
Running time: 1.942e-06 seconds
Run number: 8
Running time: 1.941e-06 seconds
Run number: 9
Running time: 1.938e-06 seconds
Run number: 10
Running time: 1.935e-06 seconds
Average time: 2.4195e-06
TotalTime: 2.4195e-05
-----
```

Figure 33 - dot Product for 2^9 on Linux

```
Vector size: 1024
Run number: 1
Running time: 3.921e-06 seconds
Run number: 2
Running time: 3.945e-06 seconds
Run number: 3
Running time: 3.881e-06 seconds
Run number: 4
Running time: 3.872e-06 seconds
Run number: 5
Running time: 3.876e-06 seconds
Run number: 6
Running time: 3.87e-06 seconds
Run number: 7
Running time: 4.049e-06 seconds
Run number: 8
Running time: 3.865e-06 seconds
Run number: 9
Running time: 3.875e-06 seconds
Run number: 10
Running time: 3.874e-06 seconds
Average time: 3.9028e-06
TotalTime: 3.9028e-05
-----
```

Figure 34 - dot Product for 2^{10} on Linux

```
Vector size: 2048
Run number: 1
Running time: 7.795e-06 seconds
Run number: 2
Running time: 7.722e-06 seconds
Run number: 3
Running time: 7.682e-06 seconds
Run number: 4
Running time: 7.765e-06 seconds
Run number: 5
Running time: 7.882e-06 seconds
Run number: 6
Running time: 7.699e-06 seconds
Run number: 7
Running time: 7.984e-06 seconds
Run number: 8
Running time: 7.707e-06 seconds
Run number: 9
Running time: 7.754e-06 seconds
Run number: 10
Running time: 7.909e-06 seconds
Average time: 7.7899e-06
TotalTime: 7.7899e-05
-----
```

Figure 35 - dot Product for 2^{11} on Linux

```

Vector size: 4096
Run number: 1
Running time: 1.5552e-05 seconds
Run number: 2
Running time: 1.9993e-05 seconds
Run number: 3
Running time: 1.6592e-05 seconds
Run number: 4
Running time: 1.5447e-05 seconds
Run number: 5
Running time: 1.5545e-05 seconds
Run number: 6
Running time: 1.5489e-05 seconds
Run number: 7
Running time: 1.5782e-05 seconds
Run number: 8
Running time: 1.5592e-05 seconds
Run number: 9
Running time: 1.5703e-05 seconds
Run number: 10
Running time: 1.5725e-05 seconds
Average time: 1.6142e-05
TotalTime: 0.00016142
-----
```

Figure 36 - dot Product for 2^{12} on Linux

```

Vector size: 8192
Run number: 1
Running time: 3.1022e-05 seconds
Run number: 2
Running time: 3.0947e-05 seconds
Run number: 3
Running time: 3.1155e-05 seconds
Run number: 4
Running time: 3.1177e-05 seconds
Run number: 5
Running time: 3.095e-05 seconds
Run number: 6
Running time: 3.1013e-05 seconds
Run number: 7
Running time: 3.3696e-05 seconds
Run number: 8
Running time: 3.1021e-05 seconds
Run number: 9
Running time: 3.0846e-05 seconds
Run number: 10
Running time: 3.1056e-05 seconds
Average time: 3.12883e-05
TotalTime: 0.000312883
-----
```

Figure 37 - dot Product for 2^{13} on Linux

```

Vector size: 16384
Run number: 1
Running time: 6.1813e-05 seconds
Run number: 2
Running time: 6.1885e-05 seconds
Run number: 3
Running time: 6.1567e-05 seconds
Run number: 4
Running time: 6.2079e-05 seconds
Run number: 5
Running time: 6.2314e-05 seconds
Run number: 6
Running time: 6.2128e-05 seconds
Run number: 7
Running time: 6.2204e-05 seconds
Run number: 8
Running time: 7.2557e-05 seconds
Run number: 9
Running time: 8.3255e-05 seconds
Run number: 10
Running time: 7.4964e-05 seconds
Average time: 6.64766e-05
TotalTime: 0.000664766
-----
```

Figure 38 - dot Product for 2^{14} on Linux

```
Vector size: 32768
Run number: 1
Running time: 0.000159884 seconds
Run number: 2
Running time: 0.000212266 seconds
Run number: 3
Running time: 0.000221654 seconds
Run number: 4
Running time: 0.000216235 seconds
Run number: 5
Running time: 0.000216026 seconds
Run number: 6
Running time: 0.000275502 seconds
Run number: 7
Running time: 0.000220655 seconds
Run number: 8
Running time: 0.000215623 seconds
Run number: 9
Running time: 0.000260834 seconds
Run number: 10
Running time: 0.000276279 seconds
Average time: 0.000227496
TotalTime: 0.00227496
```

Figure 39 - dot Product for 2^{15} on Linux

```
Vector size: 65536
Run number: 1
Running time: 0.000531392 seconds
Run number: 2
Running time: 0.00071186 seconds
Run number: 3
Running time: 0.000342346 seconds
Run number: 4
Running time: 0.000246193 seconds
Run number: 5
Running time: 0.000261437 seconds
Run number: 6
Running time: 0.000245665 seconds
Run number: 7
Running time: 0.000280828 seconds
Run number: 8
Running time: 0.000276728 seconds
Run number: 9
Running time: 0.000449214 seconds
Run number: 10
Running time: 0.000288167 seconds
Average time: 0.000363383
TotalTime: 0.00363383
```

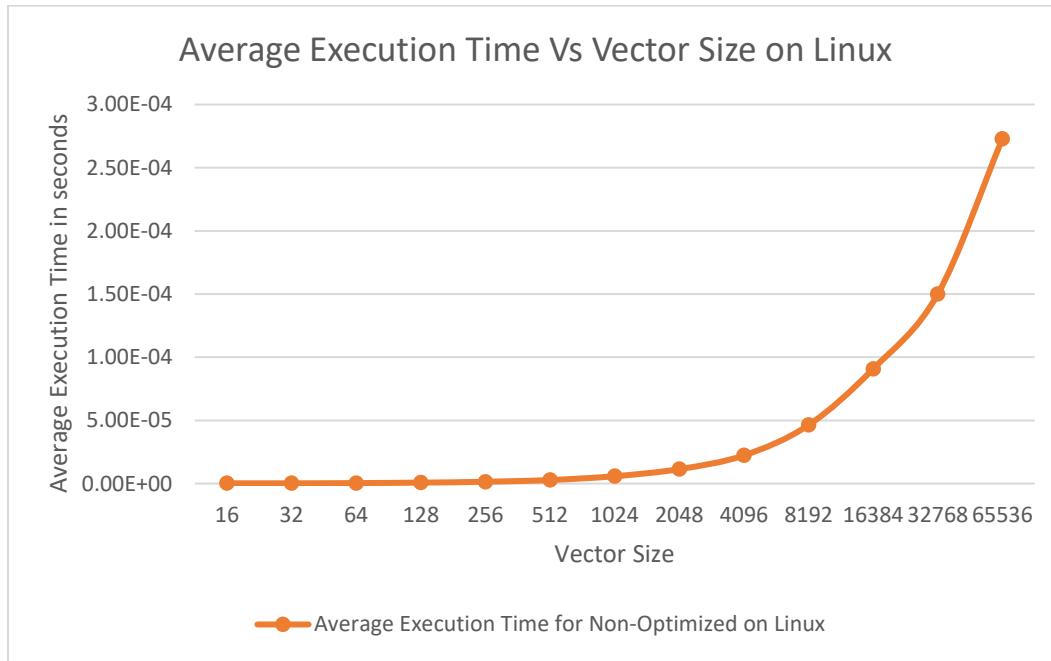
Figure 40 - dot Product for 2^{16} on Linux

Graph

Now that we have execution time, we will graph Average Execution time Vs Vector Size for unoptimized code. We ran the code 10 times for each vector size to be able to find the average execution time that is more accurate.

Table 3 - Task 2 - Linux, Ubuntu

| Vector Size | Average Execution Time in seconds for Non-Optimized on Linux |
|-------------|--|
| 16 | 2.28E-07 |
| 32 | 2.17E-07 |
| 64 | 3.97E-07 |
| 128 | 7.36E-07 |
| 256 | 1.43E-06 |
| 512 | 2.82E-06 |
| 1024 | 5.83E-06 |
| 2048 | 1.14E-05 |
| 4096 | 2.23E-05 |
| 8192 | 4.63E-05 |
| 16384 | 9.07E-05 |
| 32768 | 1.50E-04 |
| 65536 | 2.73E-04 |



Graph 2 - Average Execution Time Vs Vector Size for Non-Optimized on Linux

Task 3 - Compute dot product Using C++ (Non-Optimized and Optimized)

Windows – Intel x64

Non-Optimized Assembly Code

The Automatic Parallelization, Automatic Vectorization, and optimization from previous task is still disable. To get the non-Optimized assembly code, we can go to the project properties. In order to get to the project properties, we need to right-click on our project name.

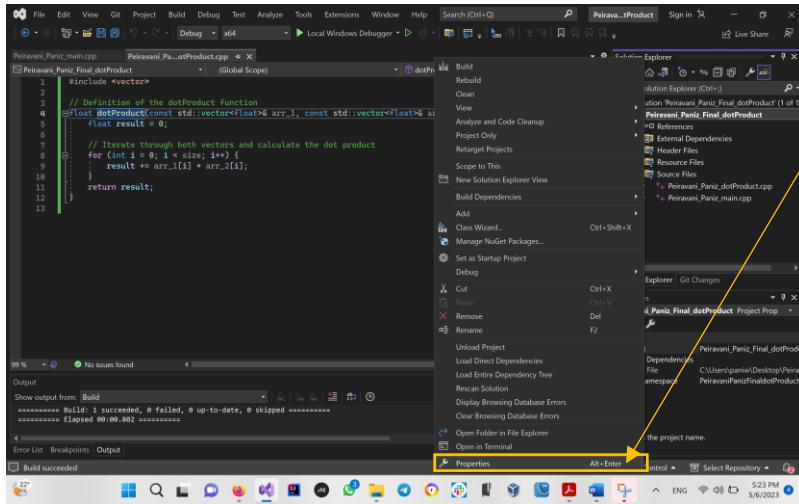


Figure 41 - Project Properties on VS – windows

Once we are in project properties, we can use the steps below to see the assembly code (.asm).

Project Property, C/C++, Output Files, and set the assembly output to *Assembly with Source Code (/FAs)*. The */FAs* will give us the assembly code, *.asm*. If we want *assembly, machine language, and source code* we can use */FACs* which will give us *.dot*.

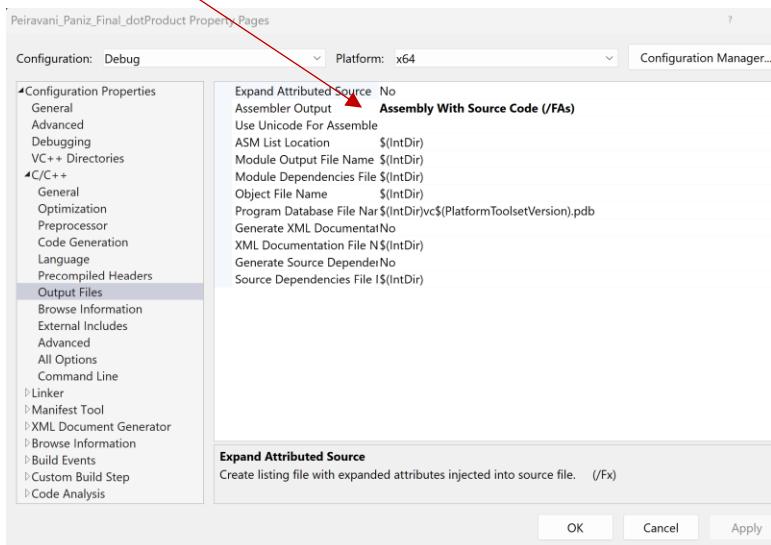


Figure 42 - Assembly Output on windows

Once we build our code, we can go to the project folder and we will see .asm output.

```

8408     main    PROC
8409
8410     ; 14 : int main() {
8411
8412     $LN15:
8413         push    rbp
8414         push    rsi
8415         push    rdi
8416         sub     rsp, 1248          ; 000004e0H
8417         lea     rbp, QWORD PTR [rsp+32]
8418         lea     rdi, QWORD PTR [rsp+32]
8419         mov     ecx, 208           ; 000000d0H
8420         mov     eax, -858993460      ; cccccccch
8421         rep     stosd
8422         mov     rax, QWORD PTR __security_cookie
8423         xor     rax, rbp
8424         mov     QWORD PTR __ArrayPad5[rbp], rax
8425         lea     rcx, OFFSET FLAT:_6AFB9823_Peiravani_Paniz_main@cpp
8426         call    __CheckForDebuggerJustMyCode
8427
8428     ; 15 :     _int64 ctr_1 = 0, ctr_2 = 0, freq = 0;
8429
8430         mov     QWORD PTR ctr_1$[rbp], 0
8431         mov     QWORD PTR ctr_2$[rbp], 0
8432         mov     QWORD PTR freq$[rbp], 0
8433
8434     ; 16 :     int acc = 0, n = 0, d_prod = 0;
8435
8436         mov     DWORD PTR acc$[rbp], 0
8437         mov     DWORD PTR n$[rbp], 0
8438         mov     DWORD PTR d_prod$[rbp], 0

```

Figure 43 - .asm file for main.cpp - Non-Optimized

```

184     ; Function compile flags: /Odp /RTCsu /ZI
185     ; File C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\14.34.31933\include\vector
186     ; COMDAT ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEEM_K@Z
187     _TEXT    SEGMENT
188     _My_data$ = 8
189     this$ = 256
190     _Pos$ = 264
191     ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEEM_K@Z PROC ; std::vector<float, std::allocator<float> >::operator[]
192
193     ; 1955 :     _NODISCARD _CONSTEXPR20 const _Ty& operator[](const size_type _Pos) const noexcept /* strengthened
194
195     $LN12:
196         mov     QWORD PTR [rsp+16], rdx
197         mov     QWORD PTR [rsp+8], rcx
198         push    rbp
199         push    rdi
200         sub     rsp, 280            ; 00000118H
201         lea     rbp, QWORD PTR [rsp+48]
202         lea     rcx, OFFSET FLAT:_FE5E3416_vector
203         call    __CheckForDebuggerJustMyCode
204
205     ; 1956 :     auto& _My_data = _Mypair._Myval2;
206
207         mov     rax, QWORD PTR this$[rbp]
208         mov     QWORD PTR _My_data$[rbp], rax
209     $LN4@operator:
210
211     ; 1957 : #if _CONTAINER_DEBUG_LEVEL > 0
212     ; 1958 :     _STL_VERIFY(
213
214         mov     rax, QWORD PTR _My_data$[rbp]
215         mov     rcx, QWORD PTR _My_data$[rbp]

```

Figure 44 - .asm file for dotProduct.cpp - non-Optimized

It will give us 11170 lines of code for main.cpp and 351 line of code for dotProduct.cpp when our code is not optimized.

```

11161      lea rcx, OFFSET FLAT:_6AFB9823_Peiravani_Paniz_main@cpp
11162      call  __CheckForDebuggerJustMyCode
11163      lea rsp, QWORD PTR [rbp+200]
11164      pop rdi
11165      pop rbp
11166      ret 0
11167      ?__empty_global_delete@@YAXPEAX@Z ENDP      ; __empty_global_delete
11168      TEXT    ENDS
11169      END
11170

```

Figure 45 - asm file for dotProduct.cpp - non-Optimized for main.cpp

```

343      lea rbp, QWORD PTR [rbp+200]
344      pop rdi
345      pop rbp
346      ret 0
347      ?dotProduct@YAMAEBV?$vector@MV?$allocator@M@std@@@std@@@H@Z ENDP ; dotProduct
348      TEXT    ENDS
349      END
350
351
352

```

Figure 46 - asm file for dotProduct.cpp - non-Optimized for dotProduct

Non-Optimized complete Assembly Code

Below we can see the complete non-optimized assembly Code.

```

282 ; 4      : float dotProduct(const std::vector<float>& arr_1, const std::vector<float>& arr_2, const int size) {
283
284     $LN6:
285     mov DWORD PTR [rsp+24], r8d
286     mov QWORD PTR [rsp+16], rdx
287     mov QWORD PTR [rsp+8], rcx
288     push rbp
289     push rdi
290     sub rsp, 312           ; 00000138H
291     lea rbp, QWORD PTR [rsp+48]
292     lea rcx, OFFSET FLAT:_D0D3A000_Peiravani_Paniz_dotProduct@cpp
293     call  __CheckForDebuggerJustMyCode
294
295 ; 5      :     float result = 0;
296
297     xorps  xmm0, xmm0
298     movss  DWORD PTR result$[rbp], xmm0
299
300 ; 6      :
301 ; 7      : // Iterate through both vectors and calculate the dot product
302 ; 8      :     for (int i = 0; i < size; i++) {
303
304     mov DWORD PTR i$1[rbp], 0
305     jmp SHORT $LN4@dotProduct
306     $LN2@dotProduct:
307     mov eax, DWORD PTR i$1[rbp]
308     inc eax
309     mov DWORD PTR i$1[rbp], eax
310     $LN4@dotProduct:
311     mov eax, DWORD PTR size$[rbp]
312     cmp DWORD PTR i$1[rbp], eax
313     jge SHORT $LN3@dotProduct
314
315 ; 9      :     result += arr_1[i] * arr_2[i];
316

```

Figure 47 - O1 Assembly Code on Windows

```

315 ; 9 :      result += arr_1[i] * arr_2[i];
316     movsx rax, DWORD PTR i$1[rbp]
317     mov rdx, rax
318     mov rcx, QWORD PTR arr_1$[rbp]
319     call ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEAE_K@Z ; std::vector<float, std::allocator<float>>::operator[]
320     mov QWORD PTR tv67[rbp], rax
321     movsx rdx, DWORD PTR i$1[rbp]
322     mov rcx, QWORD PTR arr_2$[rbp]
323     mov rdx, rcx
324     mov rcx, QWORD PTR arr_2$[rbp]
325     call ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEAE_K@Z ; std::vector<float, std::allocator<float>>::operator[]
326     mov rcx, QWORD PTR tv67[rbp]
327     movss xmm0, DWORD PTR [rcx]
328     mulss xmm0, DWORD PTR [rax]
329     movss xmm1, DWORD PTR result$[rbp]
330     addss xmm1, xmm0
331     movaps xmm0, xmm1
332     movss DWORD PTR result$[rbp], xmm0
333
334 ; 10 : }
335
336 jmp SHORT $LN2@dotProduct
$LN3@dotProduct:
337
338 ; 11 : return result;
339     movss xmm0, DWORD PTR result$[rbp]
340
341 ; 12 : }
342
343     lea rsp, QWORD PTR [rbp+264]
344     pop rdi
345     pop rbp
346     ret 0
347 ?dotProduct@@YAAEAEV?$vector@MV?$allocator@M@std@@@std@@@0H@Z ENDP ; dotProduct
350 _TEXT ENDS

```

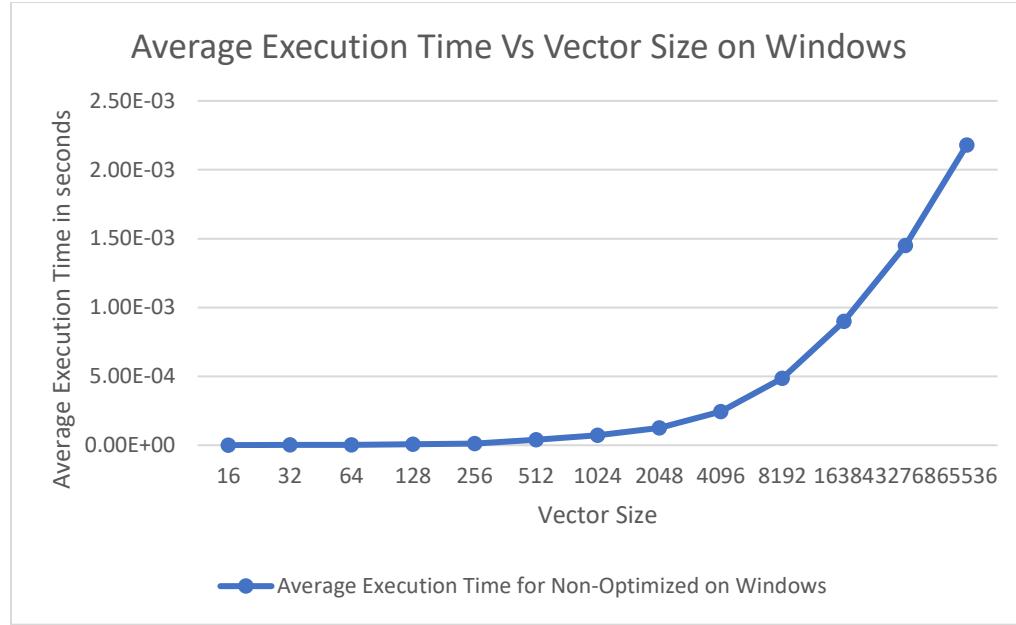
99% ▾ No issues found

Figure 48 - O1 Assembly Code on Windows

Graph

Table 4 - Non-Optimized Execution in second for windows

| Vector Size | Execution time for Non-Optimized on Windows |
|-------------|---|
| 16 | 8.20E-07 |
| 32 | 2.04E-06 |
| 64 | 3.84E-06 |
| 128 | 6.71E-06 |
| 256 | 1.28E-05 |
| 512 | 4.05E-05 |
| 1024 | 7.33E-05 |
| 2048 | 1.25E-04 |
| 4096 | 2.44E-04 |
| 8192 | 4.87E-04 |
| 16384 | 9.00E-04 |
| 32768 | 1.45E-03 |
| 65536 | 2.18E-03 |



Graph 3 - Non-Optimized Execution in second for windows

Compiler Optimization

First, we need to enable Automatic Parallelization, /Qpar, and Automatic Vectorization, /arch. We can enable them like how we disable them in task 2. To enable automatic parallelization and automatic vectorization, we need to go to the project properties. In order to get to the project properties, we need to right-click on our project name.

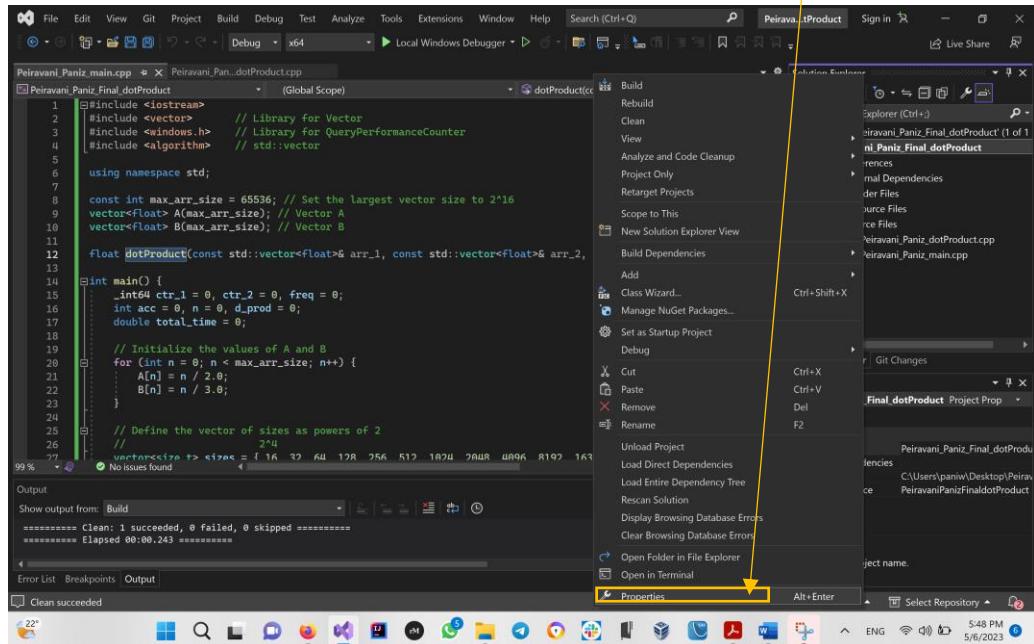


Figure 49 – Project Properties on VS – windows

Once we are in project properties, we can use the steps below to enable Parallel Code Generation and Enhanced Instruction Set.

Project Property, C/C++, Code Generation, and enable Parallel Code Generation by using *Yes (/Qpar)* and enable Enhanced Instruction Set by using *Yes Enhanced Instructions (/arch:AVX)* and then press apply. Based on our task 1, I know that my hardware can handle AVX.

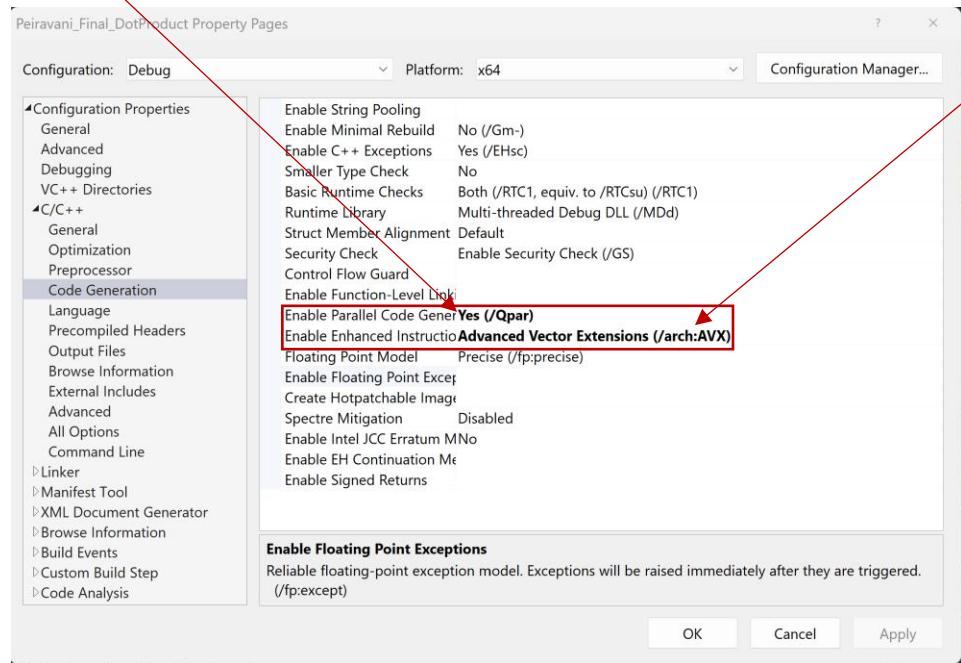


Figure 50 - enable Parallel Code Generation and Enhanced Instruction Set on VS – Windows

And choose *optimization* on optimization section on Visual Studio. We will one time choose O1 which is good for size and O2 which is good for speed.

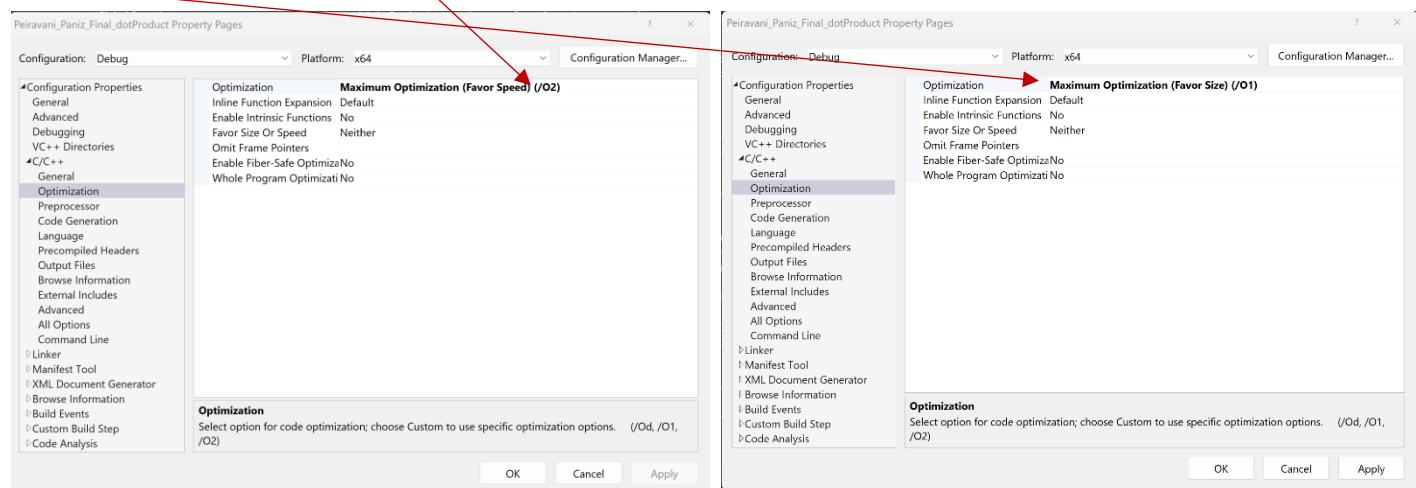


Figure 51 – Compiler Optimization on VS

01 - Optimization

Once we build our code, we can go to the project folder, and we will see .asm output which is optimized for O1.

Peiravani_Paniz_main.asm -> x Peiravani_Paniz_main.cpp Peiravani_Paniz...dotProduct.cpp

```
5981 ; File C:\Users\paniz\Desktop\Peiravani_Paniz_Final_dotProduct\Peiravani_Paniz_Final_dotProduct\Peiravani_Paniz_Final_dotProduct\Pe
5982 ; COMDAT main
5983 _TEXT SEGMENT
5984 $T1 = 32
5985 $T2 = 48
5986 $T3 = 160
5987 $T4 = 176
5988 freq$ = 192
5989 ctr_2$ = 200
5990 ctr_1$ = 208
5991 size$ = 216
5992 __$ArrayPad$ = 248
5993 main PROC ; COMDAT
5994 ; 14 : int main() {
5995
5996
5997 $LN29:
5998     mov rax, rsp
5999     mov QWORD PTR [rax+8], rbx
6000     mov QWORD PTR [rax+16], rsi
6001     mov QWORD PTR [rax+24], rdi
6002     push rbp
6003     push r14
6004     push r15
6005     lea rbp, QWORD PTR [rax-88]
6006     sub rsp, 320 ; 00000140H
6007     movaps XMMWORD PTR [rax-40], xmm6
6008     movaps XMMWORD PTR [rax-56], xmm7
6009     movaps XMMWORD PTR [rax-72], xmm8
6010     movaps XMMWORD PTR [rax-88], xmm9
6011     mov rax, QWORD PTR __security_cookie
6012     xor rax, rax
6013
6014 ; No issues found
```

Figure 52 - O1 optimized code for main.cpp

Peiravani_Paniz_main.asm

Peiravani_Paniz_main.cpp

Peiravani_Paniz_main.cpp*

```
266 _TEXT ENDS
267 ; Function compile flags: /Ogtpy
268 ; File C:\Users\paniw\Desktop\Peiravani_Paniz_Final_dotProduct\Peiravani_Paniz_Final_dotProduct\Peiravani_Paniz_Final_dotProduct\Pe
269 ; COMDAT ?dotProduct@@YAMAEBV?$vector@MV?$allocator@M@std@@std@@0H@Z
270 _TEXT SEGMENT
271 arr_1$ = 64
272 arr_2$ = 72
273 size$ = 80
274 ?dotProduct@@YAMAEBV?$vector@MV?$allocator@M@std@@std@@0H@Z PROC ; dotProduct, COMDAT
275
276 ; 4 : float dotProduct(const std::vector<float>& arr_1, const std::vector<float>& arr_2, const int size) {
277
278 $LN12:
279     mov QWORD PTR [rsp+24], rbx
280     mov QWORD PTR [rsp+32], rbp
281     push r14
282     sub rsp, 48          ; 00000030H
283     mov r14, rcx
284     movaps XMMWORD PTR [rsp+32], xmm6
285     lea rcx, OFFSET FLAT:_D0D3A000_Peiravani_Paniz_dotProduct@cpp
286     mov ebx, r8d
287     mov rbp, rdx
288     call __CheckForDebuggerJustMyCode
289     xorps xmm6, xmm6
290     test ebx, ebx
291     jle SHORT $LN10@dotProduct
292     mov QWORD PTR [rsp+64], rsi
293     mov esi, ebx
294     mov QWORD PTR [rsp+72], rdi
295     xor edi, edi
296     npad 1
297     $LL4@dotProduct:
```

Figure 53 - O1 optimized code for dorProduct.cpp

We can see that our assembly is using vectorization since we enable it.

```

3375    226 : #endif // _HAS_CXX20
3376    227 :
3377    228 :     if (_Bytes >= _Big_allocation_threshold) { // boost the alignment of big allocations to help autovectorization
3378    229 :         return _Allocate_manually_vector_aligned<_Traits>(_Bytes);
3379
3380     mov rcx, rbx
3381

```

Figure 54 – vectorization

And our code is shorter than non-optimized. We can compare the blue box in in *Figure 45* for main.cpp and green box on *Figure 46* for dotProduct.cpp to see the difference.

```

8871  __empty_global_delete@YAXPEAX@Z PROC      ; __empty_global_delete, COMDAT
8872  ?__empty_global_delete@YAXPEAX@Z PROC      ; __empty_global_delete, COMDAT
8873      lea rcx, OFFSET FLAT:_6AFB9823_Peiravani_Paniz_main@cpp
8874      jmp __CheckForDebuggerJustMyCode
8875  ?__empty_global_delete@YAXPEAX@Z ENDP      ; __empty_global_delete
8876  _TEXT  ENDS
8877  END
8878

```

Figure 55 – O1 optimized code for main.cpp

```

337      ret 0
338  ?dotProduct@YAMAEBV?$vector@M@$allocator@M@std@@@std@@0H@Z ENDP ; dotProduct
339  _TEXT  ENDS
340  END
341

```

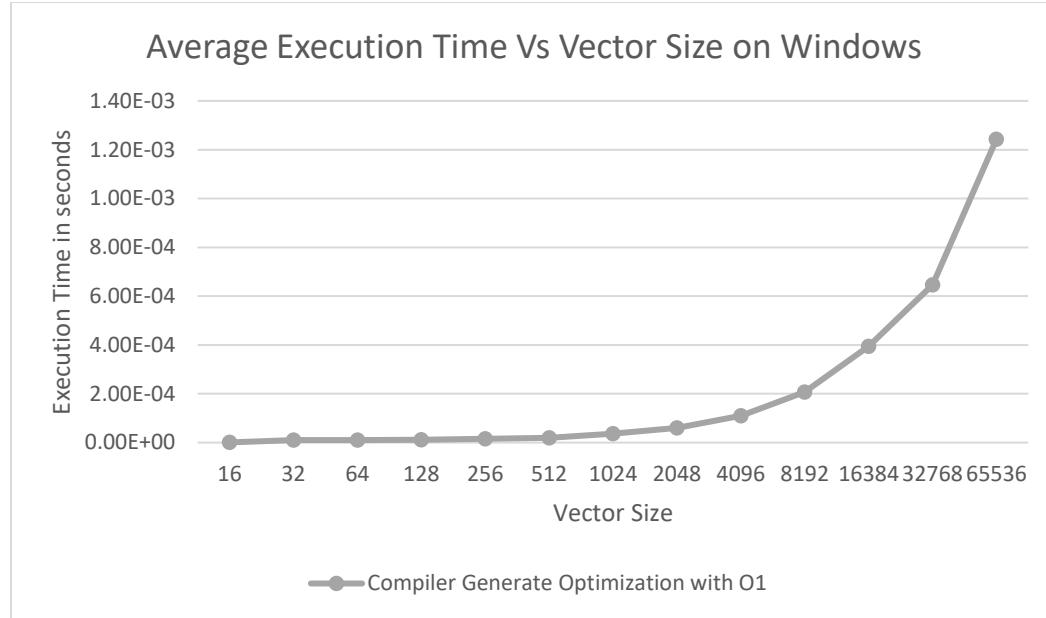
Figure 56 - O1 optimized code for dotProduct.cpp

Graph

On the Table and graph below, we can see the execution in Windows time for O1 optimization after we run the code.

Table 5 – Execution time for Compiler Generated Optimization with O1

| Vector Size | Execution time in second for Compiler Generated Optimization with O1 |
|-------------|--|
| 16 | 6.90E-07 |
| 32 | 9.97E-06 |
| 64 | 1.08E-05 |
| 128 | 1.17E-05 |
| 256 | 1.50E-05 |
| 512 | 2.00E-05 |
| 1024 | 3.64E-05 |
| 2048 | 6.06E-05 |
| 4096 | 1.10E-04 |
| 8192 | 2.07E-04 |
| 16384 | 3.95E-04 |
| 32768 | 6.46E-04 |
| 65536 | 1.24E-03 |



Graph 4 - Execution time for Compiler Generated Optimization with O1

Complete Assembly Code for O1

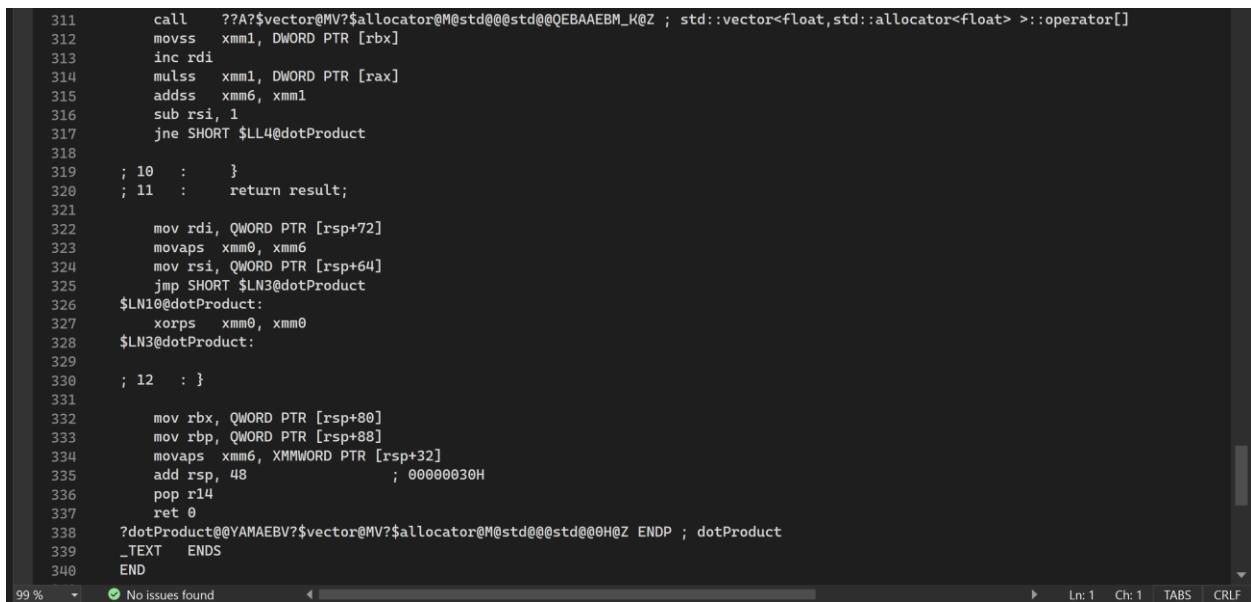
Below we can see the O1 optimization complete assembly Code in Windows Intel x64.

```

Peiravani_Pan...imization.asm  ✘ Peiravani_Paniz_main.asm  Peiravani_Paniz_main.cpp  Peiravani_Pan...dotProduct.cpp
276 ; 4 : float dotProduct(const std::vector<float>& arr_1, const std::vector<float>& arr_2, const int size) {
277
278     $LN12:
279     mov QWORD PTR [rsp+24], rbx
280     mov QWORD PTR [rsp+32], rbp
281     push    r14
282     sub    rsp, 48           ; 00000030H
283     mov    r14, rcx
284     movaps XMMWORD PTR [rsp+32], xmm6
285     lea    rcx, OFFSET FLAT:_D0D3A000_Peiravani_Paniz_dotProduct@cpp
286     mov    ebx, r8d
287     mov    rbp, rdx
288     call    __CheckForDebuggerJustMyCode
289     xorps  xmm6, xmm6
290     test   ebx, ebx
291     jle    SHORT $LN10$dotProduct
292     mov    QWORD PTR [rsp+64], rsi
293     mov    esi, ebx
294     mov    QWORD PTR [rsp+72], rdi
295     xor    edi, edi
296     npad   1
297     $LL4@dotProduct:
298
299 ; 5 :     float result = 0;
300 ; 6 :
301 ; 7 :     // Iterate through both vectors and calculate the dot product
302 ; 8 :     for (int i = 0; i < size; i++) {
303 ; 9 :         result += arr_1[i] * arr_2[i];
304
305     mov    rdx, rdi
306     mov    rcx, r14
307     call    ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEEM_K@Z ; std::vector<float, std::allocator<float> >::operator[]
308     mov    rdx, rdi
309     mov    rcx, rbp
310     mov    rbx, rax
311     call    ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEEM_K@Z ; std::vector<float, std::allocator<float> >::operator[]

```

Figure 57 - Complete Assembly Code for O1



```

311     call    ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEM_K@Z ; std::vector<float, std::allocator<float> >::operator[]
312     movss  xmm1, DWORD PTR [rbx]
313     inc rdi
314     mulss  xmm1, DWORD PTR [rax]
315     addss  xmm6, xmm1
316     sub rsi, 1
317     jne SHORT $LL4@dotProduct
318
319 ; 10 : }
320 ; 11 :     return result;
321
322     mov rdi, QWORD PTR [rsp+72]
323     movaps xmm0, xmm6
324     mov rsi, QWORD PTR [rsp+64]
325     jmp SHORT $LN3@dotProduct
326 $LN10@dotProduct:
327     xorps  xmm0, xmm0
328 $LN3@dotProduct:
329
330 ; 12 : }
331
332     mov rbx, QWORD PTR [rsp+80]
333     mov rbp, QWORD PTR [rsp+88]
334     movaps xmm6, XMMWORD PTR [rsp+32]
335     add rsp, 48 ; 00000030H
336     pop r14
337     ret 0
338 ?dotProduct@@YAMAEVM?$vector@MV?$allocator@M@std@@@std@@@0H@Z ENDP ; dotProduct
339 _TEXT ENDS
340 END

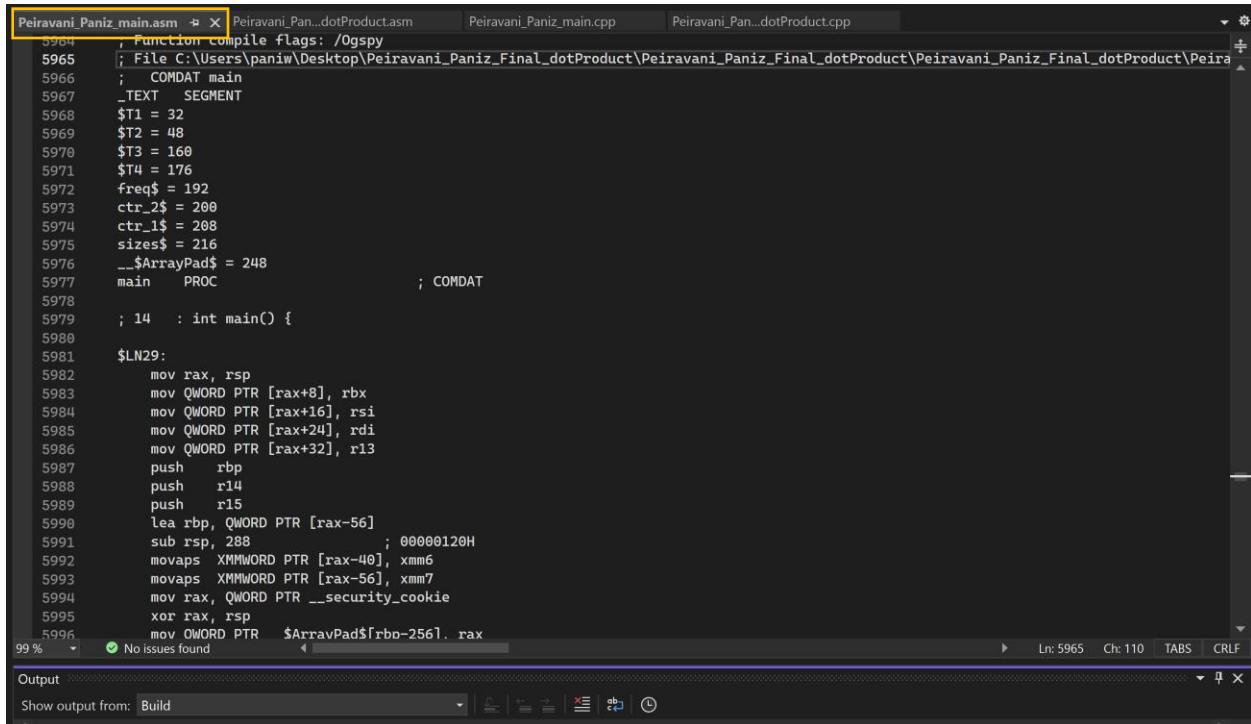
```

99 % No issues found Ln: 1 Ch: 1 TABS CRLF

Figure 58 - Complete Assembly Code for O1

O2 - Optimization

Now we can build our code for O2 to see .asm assembly code for O2.



```

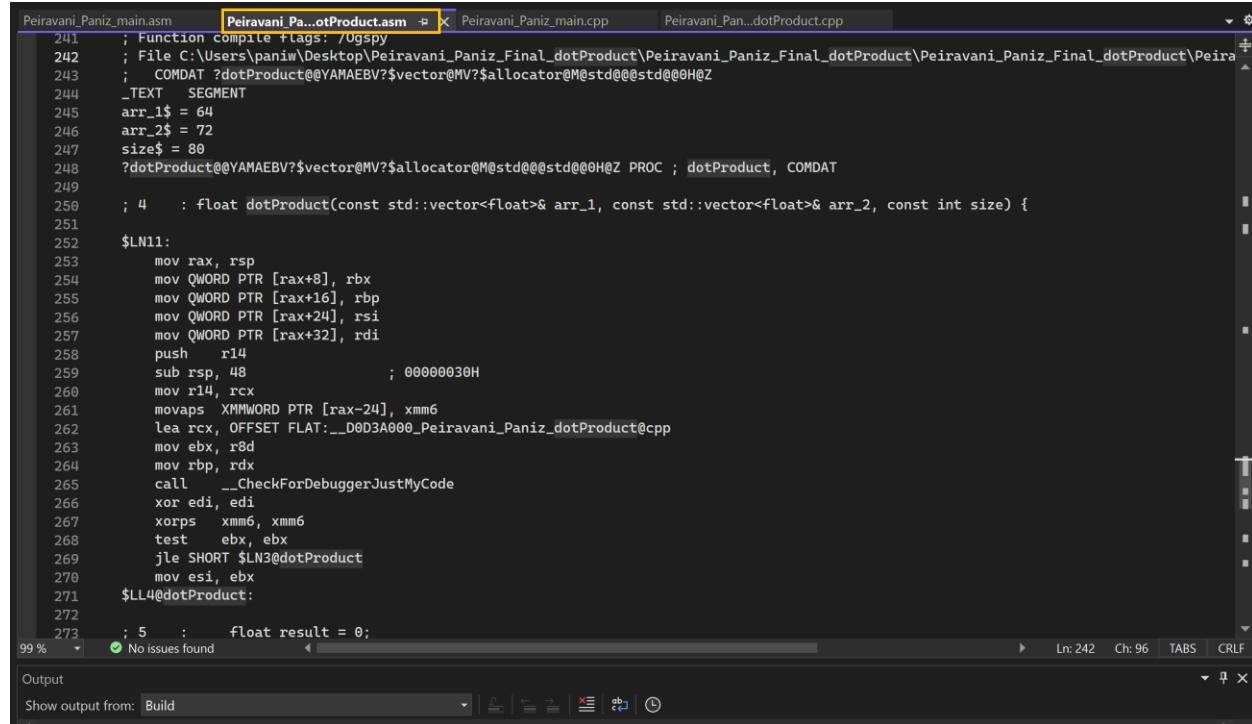
5964 ; Function compile flags: /Ogsp
5965 ; File C:\Users\paniz\Desktop\Peiravani_Paniz_Final_dotProduct\Peiravani_Paniz_Final_dotProduct\Peiravani_Paniz_Final_dotProduct\Peiravani_Paniz_main.asm
5966 ; COMDAT main
5967 _TEXT SEGMENT
5968 $T1 = 32
5969 $T2 = 48
5970 $T3 = 160
5971 $T4 = 176
5972 freq$ = 192
5973 ctr_2$ = 200
5974 ctr_1$ = 208
5975 sizes$ = 216
5976 __$ArrayPad$ = 248
5977 main PROC ; COMDAT
5978
5979 ; 14 : int main() {
5980
5981 $LN29:
5982     mov rax, rsp
5983     mov QWORD PTR [rax+8], rbx
5984     mov QWORD PTR [rax+16], rsi
5985     mov QWORD PTR [rax+24], rdi
5986     mov QWORD PTR [rax+32], r13
5987     push rbp
5988     push r14
5989     push r15
5990     lea rbp, QWORD PTR [rax-56]
5991     sub rsp, 288 ; 00000120H
5992     movaps XMMWORD PTR [rax-40], xmm6
5993     movaps XMMWORD PTR [rax-56], xmm7
5994     mov rax, QWORD PTR __security_cookie
5995     xor rax, rsp
5996     mov QWORD PTR __$ArrayPad$[rbx-256].rax

```

99 % No issues found Ln: 5965 Ch: 110 TABS CRLF

Output
Show output from: Build

Figure 59 – O2 optimized code for main.cpp



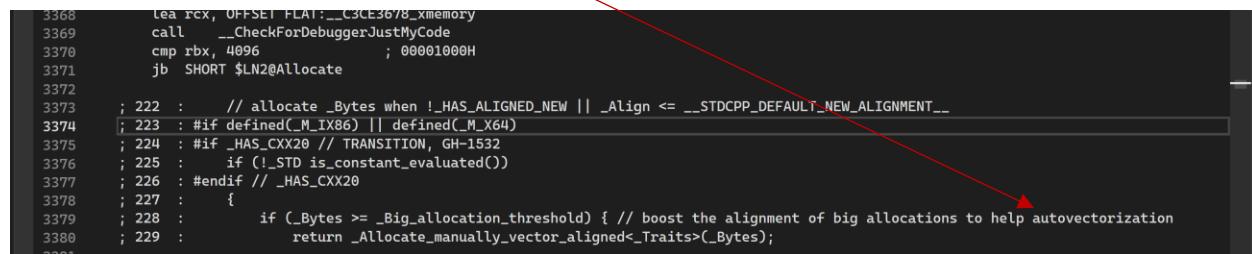
```

241 ; Function compile flags: /Ogspy
242 ; File C:\Users\paniz\Desktop\Peiravani_Paniz_Final_dotProduct\Peiravani_Paniz_Final_dotProduct\Peiravani_Paniz_Final_dotProduct\Peira
243 ; COMDAT ?dotProduct@@YAMAEV?$vector@M@std@@@std@@0H@Z
244 _TEXT SEGMENT
245 arr_1$ = 64
246 arr_2$ = 72
247 size$ = 80
248 ?dotProduct@@YAMAEV?$vector@M@std@@@std@@0H@Z PROC ; dotProduct, COMDAT
249 ; 4 : float dotProduct(const std::vector<float>& arr_1, const std::vector<float>& arr_2, const int size) {
250
251 $LN11:
252     mov rax, rsp
253     mov QWORD PTR [rax+8], rbx
254     mov QWORD PTR [rax+16], rbp
255     mov QWORD PTR [rax+24], rsi
256     mov QWORD PTR [rax+32], rdi
257     push r14
258     sub rsp, 48           ; 00000030H
259     mov r14, rcx
260     movaps XMMWORD PTR [rax-24], xmm6
261     lea rcx, OFFSET FLAT:_D0D3A000_Peiravani_Paniz_dotProduct@cpp
262     mov ebx, r8d
263     mov rbp, rdx
264     call __CheckForDebuggerJustMyCode
265     xor edi, edi
266     xorps xmm6, xmm6
267     test ebx, ebx
268     jle SHORT $LN3@dotProduct
269     mov esi, ebx
270
271 $LL4@dotProduct:
272
273 ; 5 : float result = 0;
99 %  No issues found
Ln: 242 Ch: 96 TABS CRLF
Output
Show output from: Build

```

Figure 60 – O2 optimized code for dotProduct.cpp

We can see that our assembly is using vectorization since we enable it.



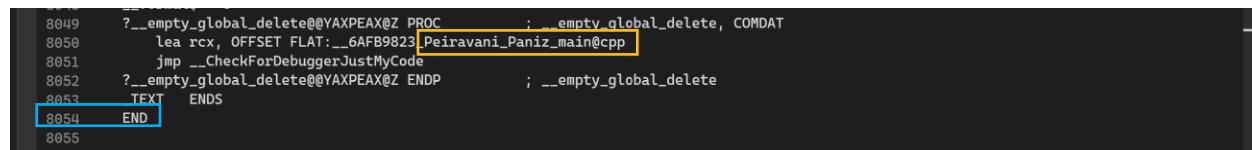
```

3368     lea rcx, OFFSET FLAT:_C3CE3678_Xmemory
3369     call __CheckForDebuggerJustMyCode
3370     cmp rbx, 4096          ; 0000100H
3371     jb SHORT $LN2@Allocate
3372
3373 ; 222 : // allocate _Bytes when !_HAS_ALIGNED_NEW || _Align <= __STDCPP_DEFAULT_NEW_ALIGNMENT__
3374 ; 223 : #if defined(_M_IX86) || defined(_M_X64)
3375 ; 224 : #if _HAS_CXX20 // TRANSITION, GH-1532
3376 ; 225 :     if (!_STD_is_constant_evaluated())
3377 ; 226 : #endif // _HAS_CXX20
3378 ; 227 :
3379 ; 228 :     if (_Bytes >= __Big_allocation_threshold) { // boost the alignment of big allocations to help autovectorization
3380 ; 229 :         return __Allocate_manually_vector_aligned<__Traits>(_Bytes);
3381

```

Figure 61 - Vectorization

And our code is shorter than non-optimized and O1 optimization. We can compare green box on *Figure 45* for main.cpp and green box on *Figure 46* for dotProduct.cpp to see the difference.



```

8049 ?__empty_global_delete@@YAXPEAX@Z PROC    ; __empty_global_delete, COMDAT
8050     lea rcx, OFFSET FLAT:_6AFB9823_Peiravani_Paniz_main@cpp
8051     jmp __CheckForDebuggerJustMyCode
8052 ?__empty_global_delete@@YAXPEAX@Z ENDP    ; __empty_global_delete
8053 _TEXT ENDS
8054 END
8055

```

Figure 62 – O2 optimized code for main.cpp



```

304     add rsp, 48           ; 00000030H
305     pop r14
306     ret 0
307 ?dotProduct@@YAMAEV?$vector@M@std@@@std@@0H@Z ENDP ; dotProduct
308 _TEXT ENDS
309 END
310

```

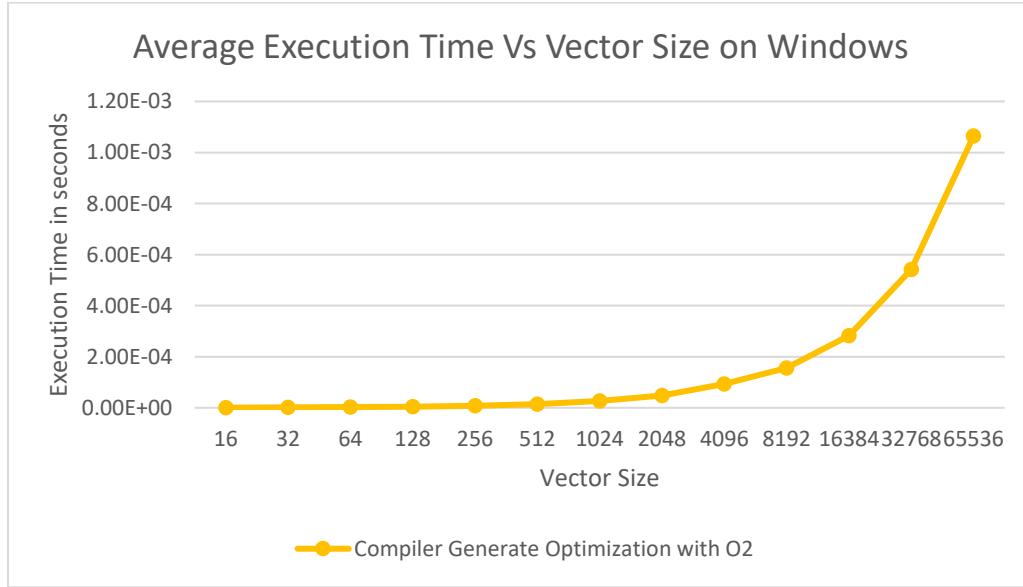
Figure 63 - O2 optimized code for dotProduct.cpp

Graph

On Table and graph below, we can see the execution time in Windows for O2 optimization after we run the code.

Table 6 - Execution time for Compiler Generated Optimization with O2

| Vector Size | Execution time in second for Compiler Generate Optimization with O2 |
|-------------|---|
| 16 | 6.60E-07 |
| 32 | 1.55E-06 |
| 64 | 2.88E-06 |
| 128 | 4.64E-06 |
| 256 | 8.25E-06 |
| 512 | 1.44E-05 |
| 1024 | 2.68E-05 |
| 2048 | 4.82E-05 |
| 4096 | 9.27E-05 |
| 8192 | 1.57E-04 |
| 16384 | 2.82E-04 |
| 32768 | 5.42E-04 |
| 65536 | 1.06E-03 |



Graph 5 - Execution time for Compiler Generated Optimization with O2

Complete O2 Optimization Code

Below we can see the O2 optimization complete assembly Code in Windows Intel x64.

```

Peiravani_Pa...otProduct.asm  Peiravani_Paniz_main.asm  Peiravani_Paniz_main.cpp  Peiravani_Pan...dotProduct.cpp
250 ; 4 : float dotProduct(const std::vector<float>& arr_1, const std::vector<float>& arr_2, const int size) {
251
252     $LN11:
253         mov rax, rsp
254         mov QWORD PTR [rax+8], rbx
255         mov QWORD PTR [rax+16], rbp
256         mov QWORD PTR [rax+24], rsi
257         mov QWORD PTR [rax+32], rdi
258         push r14
259         sub rsp, 48           ; 00000030H
260         mov r14, rcx
261         movaps XMMWORD PTR [rax-24], xmm6
262         lea rcx, OFFSET FLAT:_ZED3A000_Peiravani_Paniz_dotProduct@cpp
263         mov ebx, r8d
264         mov rbp, rdx
265         call __CheckForDebuggerJustMyCode
266         xor edi, edi
267         xorps xmm6, xmm6
268         test ebx, ebx
269         jle SHORT $LN3@dotProduct
270         mov esi, ebx
271     $LL4@dotProduct:
272
273     ; 5 :     float result = 0;
274     ; 6 :
275     ; 7 :     // Iterate through both vectors and calculate the dot product
276     ; 8 :     for (int i = 0; i < size; i++) {
277     ; 9 :         result += arr_1[i] * arr_2[i];
278
279         mov rdx, rdi
280         mov rcx, r14
281         call ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEAE@Z ; std::vector<float, std::allocator<float> >::operator[]
282         mov rdx, rdi
283         mov rcx, rbp
284         mov rbx, rax
285         call ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEAE@Z ; std::vector<float, std::allocator<float> >::operator[]

```

Figure 64 - O2 Assembly Code on Windows

```

285     call ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEAE@Z ; std::vector<float, std::allocator<float> >::operator[]
286     movss xmm1, DWORD PTR [rbx]
287     inc rdi
288     mulss xmm1, DWORD PTR [rax]
289     addss xmm6, xmm1
290     sub rsi, 1
291     jne SHORT $LL4@dotProduct
292     $LN3@dotProduct:
293
294     ; 10 : }
295     ; 11 :     return result;
296     ; 12 : }
297
298     mov rbx, QWORD PTR [rsp+64]
299     movaps xmm0, xmm6
300     movaps xmm6, XMMWORD PTR [rsp+32]
301     mov rbp, QWORD PTR [rsp+72]
302     mov rsi, QWORD PTR [rsp+88]
303     mov rdi, QWORD PTR [rsp+88]
304     add rsp, 48           ; 00000030H
305     pop r14
306     ret 0
307     ?dotProduct@YAMAEBV?$vector@MV?$allocator@M@std@@@std@@0H@Z ENDP ; dotProduct
308 _TEXT ENDS
309 END
310

```

Figure 65 - O2 Assembly Code on Windows

Manually Optimization

Now we are manually optimizing the code that we. The red lines show the code that I can remove and the green lines the code that I optimized. One way to optimize is to reduce the use of `mov` instructions by reusing registers or rearranging instructions and changing bytes storage for memory usage and speed.

```
; 4      : float dotProduct(const std::vector<float>& arr_1, const std::vector<float>& arr_2, const int size) {

$LN11:
    mov    rax, rsp

    mov    QWORD PTR [rax+8], rbx
    mov    QWORD PTR [rsp+24], rbx           //rsp = to stack pointer register and change
                                              the bytes
    mov    QWORD PTR [rax+16], rbp
    mov    QWORD PTR [rsp+32], rbp           //rsp = to stack pointer register and
                                              change bytes
    mov    QWORD PTR [rax+24], rsi
    mov    QWORD PTR [rsp+32], rdi           //rsp = to stack pointer register and
                                              change the bytes
    push   r14
    sub    rsp, 48                         ; 00000030H
    mov    r14, rcx
    movaps XMMWORD PTR [rax-24], xmm6
    movaps XMMWORD PTR [rax+32], xmm6       //rsp = to stack pointer register and
                                              change the bytes
    lea    rcx, OFFSET FLAT:_D0D3A000_Peiravani_Paniz_dotProduct@cpp
    mov    ebx, r8d
    mov    rbp, rdx
    call   __CheckForDebuggerJustMyCode
    xorps xmm6, xmm6
    test   ebx, ebx
    jle    SHORT $LN3@dotProduct
    mov    QWORD PTR [rsp+64], rsi
    mov    esi, ebx
    mov    esi, ebx
    mov    QWORD PTR [rsp+72], rdi
    xor    edi, ed
    mov    esi, ebx

$LL4@dotProduct:

; 5      :     float result = 0;
; 6      :
; 7      :     // Iterate through both vectors and calculate the dot product
; 8      :     for (int i = 0; i < size; i++) {
; 9      :         result += arr_1[i] * arr_2[i];

    mov    rdx, rdi
    mov    rcx, r14
    call   ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEAM_K@Z ;
    std::vector<float, std::allocator<float> >::operator[]
    mov    rdx, rdi
    mov    rcx, rbp
    mov    rbx, rax
    call   ??A?$vector@MV?$allocator@M@std@@@std@@QEBAEAM_K@Z ;
    std::vector<float, std::allocator<float> >::operator[]
```

```

    movss xmm1, DWORD PTR [rbx]
    inc rdi
    mulss xmm1, DWORD PTR [rax]
    addss xmm6, xmm1
    sub rsi, 1
    jne SHORT $LL4@dotProduct
$LN3@dotProduct:

; 10  :    }
; 11  :    return result;
    Mov rdi, QWORD PTR [rsp+72]
    mov rdi, QWORD PTR [rsp+72]
    mov rsi, QWORD PTR [rsp+64]
    mov rsi, QWORD PTR [rsp+80]

    vmovaps      xmm0, xmm6
    jmp     SHORT $LN3@dotProduct

$LN10@dotProduct:
    vxorps      xmm0, xmm0, xmm0

$LN3@dotProduct:
; 12  :    }

    mov rbx, QWORD PTR [rsp+64]
    mov rbx, QWORD PTR [rsp+80]
    movaps xmm0, xmm6
    movaps xmm6, XMMWORD PTR [rsp+32]
    mov rbp, QWORD PTR [rsp+72]
    mov rbx, QWORD PTR [rsp+88]
    mov rsi, QWORD PTR [rsp+80]
    mov rdi, QWORD PTR [rsp+88]
    add rsp, 48                                ; 00000030H
    pop r14
    ret 0

?dotProduct@@YMAEBV?$vector@MV?$allocator@M@std@@@std@@@0H@Z ENDP ; dotProduct
_TEXT ENDS
END

```

Now here is my final optimization code.

```

1 // Definition of the dotProduct function
2 float dotProduct_optimized(float* arr_1, float* arr_2, const int N) {
3     float result = 0.0;
4
5     // Iterate through both vectors and calculate the dot product
6     /*for (int i = 0; i < size; i++) {
7         result += arr_1[i] * arr_2[i];
8     }*/
9
10    _asm {
11        pxor xmm0, xmm0;
12
13        mov eax, dword ptr[arr_1];
14        mov ebx, dword ptr[arr_2];
15        mov ecx, dword ptr[N];
16
17        $MyLoop:
18            movups xmm0, [eax];
19            movups xmm1, [ebx];
20            mulps xmm1, xmm2;
21            addps xmm0, xmm1;
22
23            add eax, 16;
24            add ebx, 16;
25            sub ecx, 4;
26            jnz $MyLoop;
27
28            haddps xmm0, xmm0;
29            haddps xmm0, xmm0;
30            movss dword ptr[result], xmm0;
31
32        }
33
34    return result;
35 }

```

Figure 66 - Manually Optimize

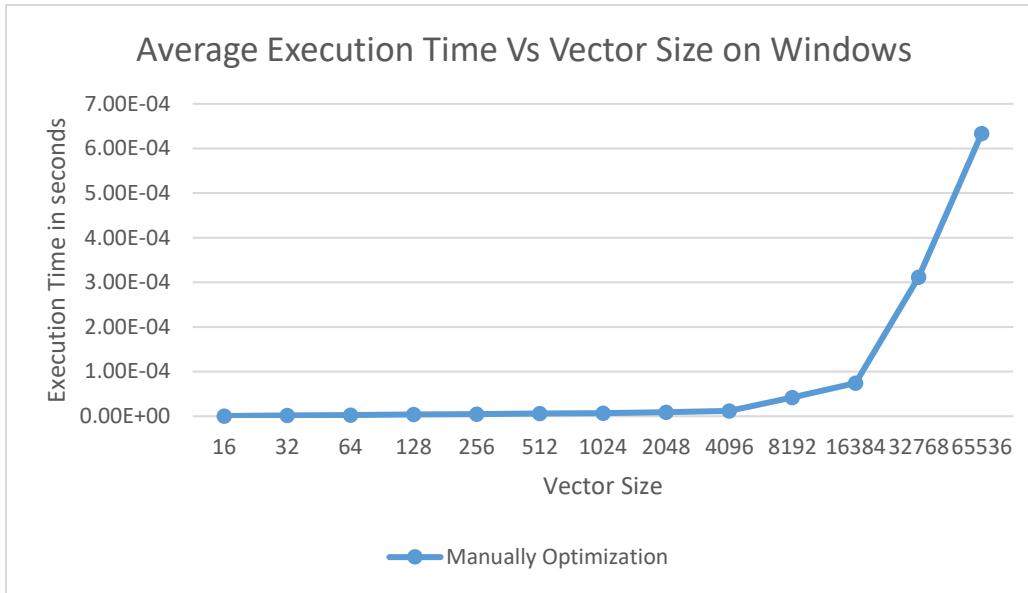
And after we run the code, we can see that the execution time is less than before. Below table shows the execution time of our manually optimization code.

Graph

On the Table and graph below, we can see the execution time in Windows for manually optimization after we run the code.

Table 7 – Execution time for Manually Optimization

| Vector Size | Execution time in second for Manually Optimization |
|-------------|--|
| 16 | 4.70E-07 |
| 32 | 1.97E-06 |
| 64 | 2.46E-06 |
| 128 | 4.19E-06 |
| 256 | 4.90E-06 |
| 512 | 6.05E-06 |
| 1024 | 7.15E-06 |
| 2048 | 9.10E-06 |
| 4096 | 1.16E-05 |
| 8192 | 4.23E-05 |
| 16384 | 7.45E-05 |
| 32768 | 3.12E-04 |
| 65536 | 6.34E-04 |

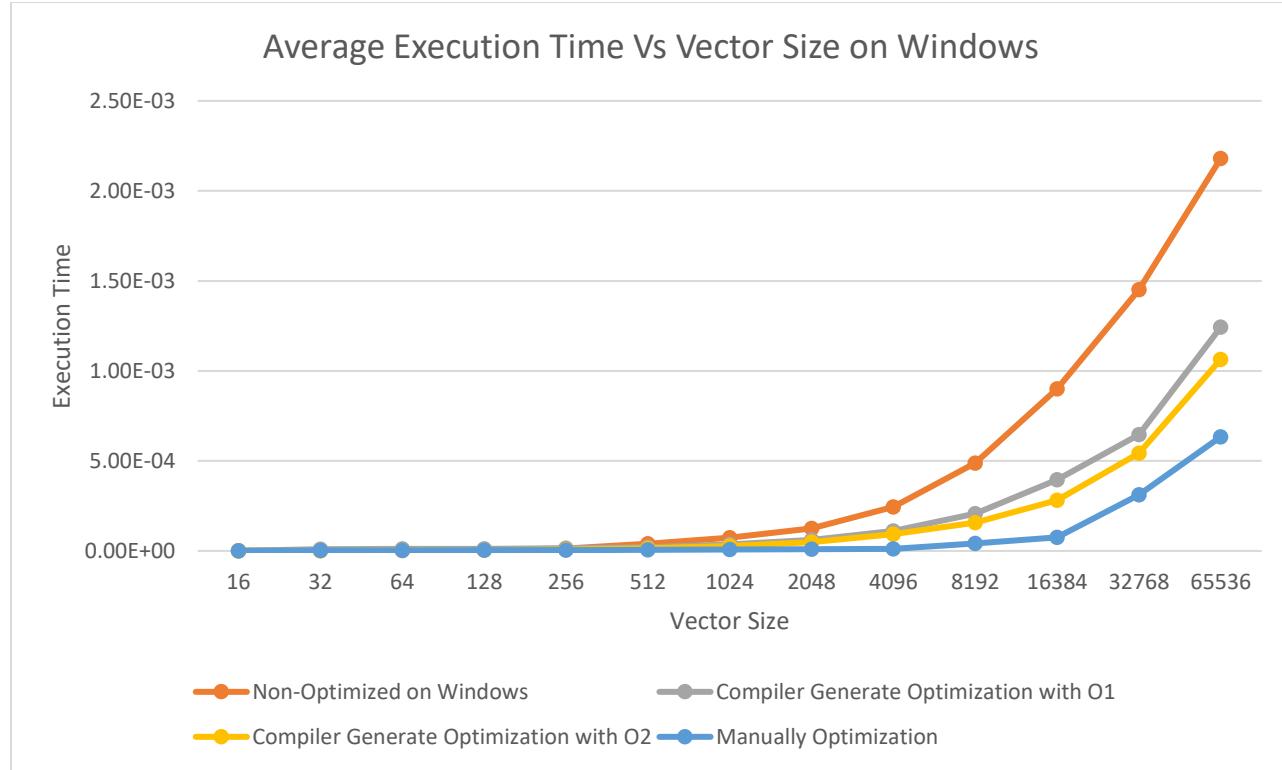


Graph

Now that we have our execution time for non-optimized, O1 optimization, O2 optimization, and manually optimization we can put the execution time on graph to compare the executions times in Windows Intel x64. We ran the code 10 times for each vector size to be able to find the average execution time that is more accurate.

Table 8 - Task 3 for Windows

| Vector Size | Non-Optimized on Windows | Compiler Generate Optimization with O1 | Compiler Generate Optimization with O2 | Manually Optimization |
|-------------|--------------------------|--|--|-----------------------|
| 16 | 8.20E-07 | 6.90E-07 | 6.60E-07 | 4.70E-07 |
| 32 | 2.04E-06 | 9.97E-06 | 1.55E-06 | 1.97E-06 |
| 64 | 3.84E-06 | 1.08E-05 | 2.88E-06 | 2.46E-06 |
| 128 | 6.71E-06 | 1.17E-05 | 4.64E-06 | 4.19E-06 |
| 256 | 1.28E-05 | 1.50E-05 | 8.25E-06 | 4.90E-06 |
| 512 | 4.05E-05 | 2.00E-05 | 1.44E-05 | 6.05E-06 |
| 1024 | 7.33E-05 | 3.64E-05 | 2.68E-05 | 7.15E-06 |
| 2048 | 1.25E-04 | 6.06E-05 | 4.82E-05 | 9.10E-06 |
| 4096 | 2.44E-04 | 1.10E-04 | 9.27E-05 | 1.16E-05 |
| 8192 | 4.87E-04 | 2.07E-04 | 1.57E-04 | 4.23E-05 |
| 16384 | 9.00E-04 | 3.95E-04 | 2.82E-04 | 7.45E-05 |
| 32768 | 1.45E-03 | 6.46E-04 | 5.42E-04 | 3.12E-04 |
| 65536 | 2.18E-03 | 1.24E-03 | 1.06E-03 | 6.34E-04 |



Graph 6 - Task 3 for windows

Linux – Ubuntu

Non-Optimized Assembly Code

The Automatic Parallelization, Automatic Vectorization, and optimization from previous task 2 still disable. To get the Non-Optimized assembly code, we can use the below command line on Ubuntu gcc. We will use the disabled flags for Automatic Parallelization, Automatic Vectorization just to be sure.

```
// The -fno-tree-vectorize flag disables automatic vectorization.
// The -fno-tree-loop-distribute-patterns flag disables automatic loop
// The -O0 will make sure that the code stays unoptimized
// -S will create a file for the assembly code

g++ -O0 -fno-tree-loop-distribute-patterns -fno-tree-vectorize
Peiravani_Paniz_dotProduct.cpp -o Peiravani_Paniz_dotProduct_NonOpt -S
```

```
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ g++ -O0 -fno-tree-loop-distribute-patterns -fno-tree-vectorize
Peiravani_Paniz_dotProduct.cpp -o Peiravani_Paniz_dotProduct_NonOpt -S
```

Figure 67 – Non-Optimized Assembly code

Complete Non-Optimized Assembly Code

Now terminal will make a file in our project folder that contains non-optimized assembly code for our C++ code.

```

1 | .file   "Peiravani_Paniz_dotProduct.cpp"
2 | .text
3 | .p2align 4
4 | .globl _Z10dotProductRKSt6vectorIfSaIfEES3_i
5 | .type   _Z10dotProductRKSt6vectorIfSaIfEES3_i, @function
6 | _Z10dotProductRKSt6vectorIfSaIfEES3_i:
7 | .LFB865:
8 |     .cfi_startproc
9 |     endbr64
10 |    testl %edx, %edx
11 |    jle   .L10
12 |    leal   -1(%rdx), %eax
13 |    movq   (%rdi), %rdi
14 |    movq   (%rsi), %rcx
15 |    cmpl   $6, %eax
16 |    jbe   .L11
17 |    movl   %edx, %esi
18 |    xorl   %eax, %eax
19 |    vxorps %xmm0, %xmm0, %xmm0
20 |    shrq   $3, %est
21 |    salq   $5, %rsi
22 |    .p2align 4,,10
23 |    .p2align 3
24 | .L4:
25 |    vmovups (%rdi,%rax), %ymm4
26 |    vmulps (%rcx,%rax), %ymm4, %ymm1
27 |    addq   $32, %rax
28 |    vaddss %xmm1, %xmm0, %xmm0
29 |    vshufps $85, %xmm1, %xmm1, %xmm3
30 |    vshufps $255, %xmm1, %xmm1, %xmm2
31 |    vaddss %xmm3, %xmm0, %xmm0
32 |    vunpckhps %xmm1, %xmm1. %xmm1. %xmm3

```

Figure 68 - Non-Optimized on Linux

```

33 |    vextractf128 $0x1, %ymm1, %xmm1
34 |    vaddss %xmm3, %xmm0, %xmm0
35 |    vaddss %xmm2, %xmm0, %xmm0
36 |    vshufps $85, %xmm1, %xmm1, %xmm2
37 |    vaddss %xmm1, %xmm0, %xmm0
38 |    vaddss %xmm2, %xmm0, %xmm0
39 |    vunpckhps %xmm1, %xmm1, %xmm2
40 |    vshufps $255, %xmm1, %xmm1, %xmm1
41 |    vaddss %xmm2, %xmm0, %xmm0
42 |    vaddss %xmm1, %xmm0, %xmm0
43 |    cmpq   %rst, %rax
44 |    jne   .L4
45 |    movl   %edx, %esi
46 |    andl   $-8, %esi
47 |    movl   %esi, %eax
48 |    cmpl   %esi, %edx
49 |    je    .L16
50 |    vzeroupper
51 | .L3:
52 |    movl   %edx, %r8d
53 |    subl   %esi, %r8d
54 |    leal   -1(%r8), %r9d
55 |    cmpl   $2, %r9d
56 |    jbe   .L7
57 |    vmovups (%rdi,%rsi,4), %xmm5
58 |    vmulps (%rcx,%rst,4), %xmm5, %xmm1
59 |    movl   %r8d, %esi
60 |    andl   $-4, %esi
61 |    addl   %esi, %eax
62 |    vaddss %xmm1, %xmm0, %xmm0
63 |    vshufps $85, %xmm1, %xmm1, %xmm2
64 |    vaddss %xmm2, %xmm0, %xmm0
65 |    vunpckhns %xmm1, %xmm1. %xmm2

```

Figure 69 - Non-Optimized on Linux

```

65    vunpckhps    %xmm1, %xmm1, %xmm2
66    vshufps $255, %xmm1, %xmm1, %xmm1
67    vaddss %xmm2, %xmm0, %xmm0
68    vaddss %xmm1, %xmm0, %xmm0
69    cmpl %esi, %r8d
70    je .L1
71 .L7:
72    cltq
73    .p2align 4,,10
74    .p2align 3
75 .L9:
76    vmovss (%rdi,%rax,4), %xmm1
77    vmulss (%rcx,%rax,4), %xmm1, %xmm1
78    addq $1, %rax
79    vaddss %xmm1, %xmm0, %xmm0
80    cmpl %eax, %edx
81    jg .L9
82    ret
83    .p2align 4,,10
84    .p2align 3
85 .L10:
86    vxorps %xmm0, %xmm0, %xmm0
87 .L1:
88    ret
89    .p2align 4,,10
90    .p2align 3
91 .L16:
92    vzeroupper
93    ret
94 .L11:
95    xorl %esi, %esi
96    xorl %eax, %eax
97    vxorps %xmm0, %xmm0, %xmm0

```

Figure 70 - Non-Optimized on Linux

```

96    xorl %eax, %eax
97    vxorps %xmm0, %xmm0, %xmm0
98    jmp .L3
99    .cfi_endproc
100 .LFE865:
101    .size _Z10dotProductRKSt6vectorIfSaIfEES3_i, .-_Z10dotProductRKSt6vectorIfSaIfEES3_i
102    .ident "GCC: (Ubuntu 11.3.0-1ubuntu1-22.04) 11.3.0"
103    .section .note.GNU-stack,"",@progbits
104    .section .note.gnu.property,"a"
105    .align 8
106    .long 1f - 0f
107    .long 4f - 1f
108    .long 5
109 0:
110    .string "GNU"
111 1:
112    .align 8
113    .long 0xc0000002
114    .long 3f - 2f
115 2:
116    .long 0x3
117 3:
118    .align 8
119 4:

```

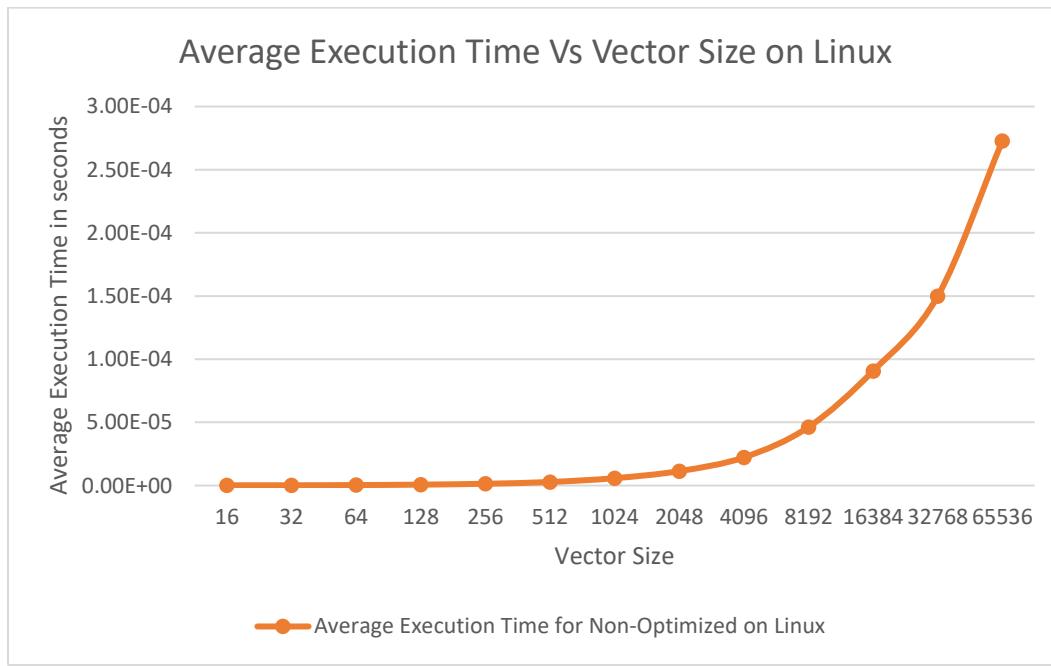
Figure 71 - Non-Optimized on Linux

Graph

On the Table and graph below, we can see the execution time in Linux for non-optimization after we run the code.

Table 9 – non-optimized assembly execution time on Linbux

| Vector Size | Execution time in seconds for Non-Optimized on Windows |
|-------------|--|
| 16 | 2.28E-07 |
| 32 | 2.17E-07 |
| 64 | 3.97E-07 |
| 128 | 7.36E-07 |
| 256 | 1.43E-06 |
| 512 | 2.82E-06 |
| 1024 | 5.83E-06 |
| 2048 | 1.14E-05 |
| 4096 | 2.23E-05 |
| 8192 | 4.63E-05 |
| 16384 | 9.07E-05 |
| 32768 | 1.50E-04 |
| 65536 | 2.73E-04 |



Graph 7 - non-optimized assembly execution time on Linbux

Compiler Optimization

O1 - Optimization

First, we need to enable Automatic Parallelization, /Qpar, and Automatic Vectorization, /arch. For enabling automatic parallelization and automatic vectorization we can use the below command line on Ubuntu gcc.

```
// The -ftree-vectorize flag eables automatic vectorization.
// The -ftree-loop-distribute-patterns flag eables automatic loop parallelization.
// The -O1 has level 1 optimization which is basic
// -mavx is for AVX vector instructor
// -S will create a file for the assembly code

g++ -O1 -ftree-loop-distribute-patterns -ftree-vectorize -mavx
Peiravani_Paniz_dotProduct.cpp -o Peiravani_Paniz_dotProduct_Opt01 -S
```

```
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ g++ -O1 -ftree-loop-distribute-patterns
-ftree-vectorize -mavx Peiravani_Paniz_dotProduct.cpp -o Peiravani_Paniz_dotProduct_Opt01 -S
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ █
```

Figure 72 – get Assembly O1 optimize on Linux

After we run the O1 optimized code, we can see the execution time is less than from non-Optimized.

```
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ g++ -O1 -ftree-loop-distribute-patterns
-ftree-vectorize -mavx Peiravani_Paniz_main.cpp Peiravani_Paniz_dotProduct.cpp -o Peiravani_Pa
niz_dotProduct_Opt01
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ ./Peiravani_Paniz_dotProduct_Opt01
█
```

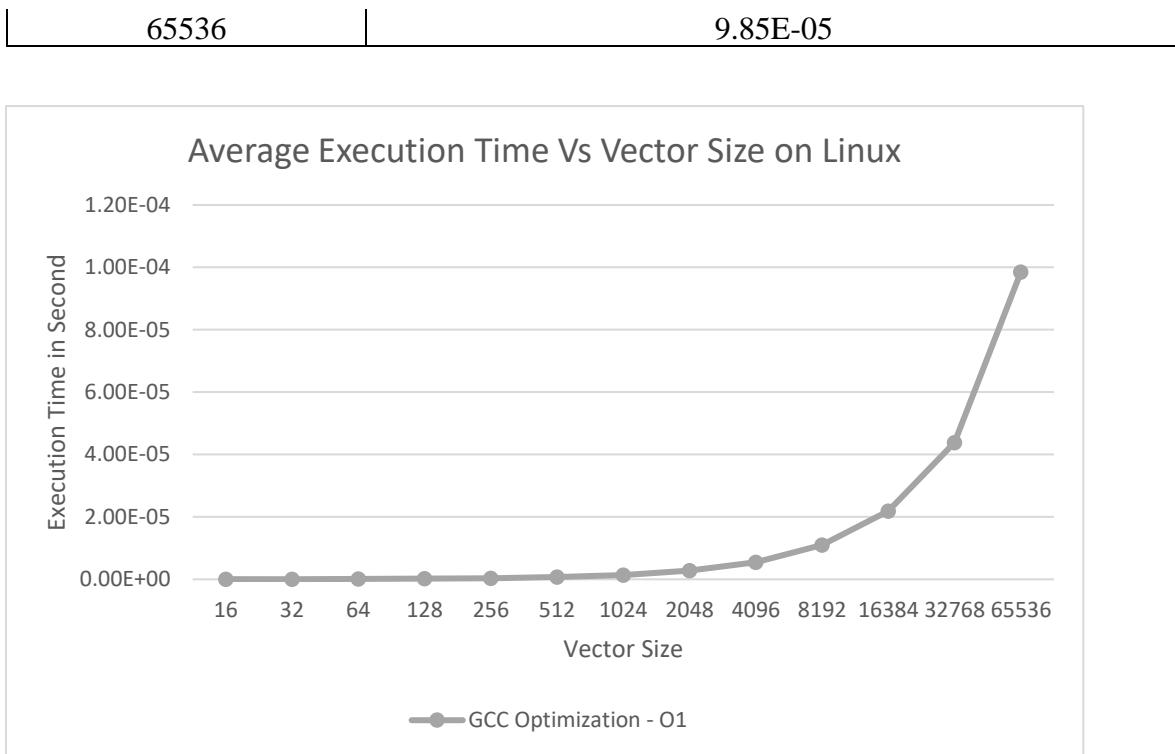
Figure 73 – run the C++ code with O1 Optimization

Graph

On the Table and graph below, we can see the execution time in Linux for O1 optimization after we run the code.

Table 10 - Execution time for Compiler Generated Optimization with O1

| Vector Size | Execution time in second for GCC Optimization with O1 |
|-------------|---|
| 16 | 5.26E-08 |
| 32 | 6.44E-08 |
| 64 | 1.08E-07 |
| 128 | 1.93E-07 |
| 256 | 3.66E-07 |
| 512 | 7.23E-07 |
| 1024 | 1.39E-06 |
| 2048 | 2.76E-06 |
| 4096 | 5.50E-06 |
| 8192 | 1.10E-05 |
| 16384 | 2.19E-05 |
| 32768 | 4.38E-05 |



Graph 8 - Execution time for Compiler Generated Optimization with O1

Complete Assembly Code for O1

Below we can see the O1 optimization complete assembly Code in Linux.

```

1 | .file  "Peiravani_Paniz_dotProduct.cpp"
2 | .text
3 | .globl _Z10dotProductRKSt6vectorIfSaIfEES3_i
4 | .type _Z10dotProductRKSt6vectorIfSaIfEES3_i, @function
5 _Z10dotProductRKSt6vectorIfSaIfEES3_i:
6 .LFB865:
7 | .cfi_startproc
8 | endbr64
9 | testl %edx, %edx
10 | jle .L8
11 | movq (%rdi), %rdi
12 | movq (%rsi), %rcx
13 | leal -1(%rdx), %eax
14 | cmpl $2, %eax
15 | jbe .L9
16 | movl %edx, %esi
17 | shrq $2, %esi
18 | movl %esi, %esi
19 | salq $4, %rsi
20 | movl $0, %eax
21 | pxor %xmm0, %xmm0
22 .L4:
23 | movups (%rdi,%rax), %xmm1
24 | movups (%rcx,%rax), %xmm3
25 | mulps %xmm3, %xmm1
26 | addss %xmm1, %xmm0
27 | movaps %xmm1, %xmm2
28 | shufps $85, %xmm1, %xmm2
29 | addss %xmm2, %xmm0
30 | movaps %xmm1, %xmm2
31 | unpckhps %xmm1, %xmm2
32 | addss %xmm2, %xmm0

```

Figure 74 - Optimized O1 on Linux

```

32      addss  %xmm2, %xmm0
33      shufps $255, %xmm1, %xmm1
34      addss  %xmm1, %xmm0
35      addq   $16, %rax
36      cmpq   %rsi, %rax
37      jne    .L4
38      movl   %edx, %esi
39      andl   $-4, %esi
40      movl   %esi, %eax
41      cmpl   %esi, %edx
42      je     .L12
43 .L3:
44      cltq
45 .L7:
46      movss  (%rdi,%rax,4), %xmm1
47      mulss  (%rcx,%rax,4), %xmm1
48      addss  %xmm1, %xmm0
49      addq   $1, %rax
50      cmpl   %eax, %edx
51      jg    .L7
52      ret
53 .L12:
54      ret
55 .L9:
56      movl   $0, %eax
57      pxor   %xmm0, %xmm0
58      jmp    .L3
59 .L8:
60      pxor   %xmm0, %xmm0
61      ret
62      .cfi_endproc
63 .LFE865:

```

Figure 75 - Optimized O1 on Linux

```

63 .LFE865:
64      .size   _Z10dotProductRKSt6vectorIfSaIfEES3_i, .-_Z10dotProductRKSt6vectorIfSaIfEES3_i
65      .ident  "GCC: (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0"
66      .section .note.GNU-stack,"",@progbits
67      .section .note.gnu.property,"a"
68      .align 8
69      .long   1f - 0f
70      .long   4f - 1f
71      .long   5
72 0:
73      .string "GNU"
74 1:
75      .align 8
76      .long   0xc0000002
77      .long   3f - 2f
78 2:
79      .long   0x3
80 3:
81      .align 8
82 4:

```

Figure 76 - Optimized O1 on Linux

O2 - Optimization

Now we are going to find the execution time using O2 optimization flags which level 2 and is more recommended. O2 more focus to optimize for speed.

```
g++ -O2 -ftree-loop-distribute-patterns -ftree-vectorize -mavx2
Peiravani_Paniz_dotProduct.cpp -o Peiravani_Paniz_dotProduct_Opt01 -S
```

```
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ g++ -O2 -ftree-loop-distribute-patterns
-ftree-vectorize -mavx Peiravani_Paniz_dotProduct.cpp -o Peiravani_Paniz_dotProduct_Opt02 -S
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$
```

Figure 77 - get Assembly O2 optimize on Linux

After we run the O2 optimized code, we can see the execution time is less than from non-Optimized.

```
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ g++ -O2 -ftree-loop-distribute-patterns -ftree-vectorize -fmax-peel Peiravani_Paniz_main.cpp Peiravani_Paniz_dotProduct.cpp -o Peiravani_Paniz_dotProduct_Opt02
paniz@paniz-VirtualBox:~/Desktop/1. Final - Part two$ ./Peiravani_Paniz_dotProduct_Opt02
```

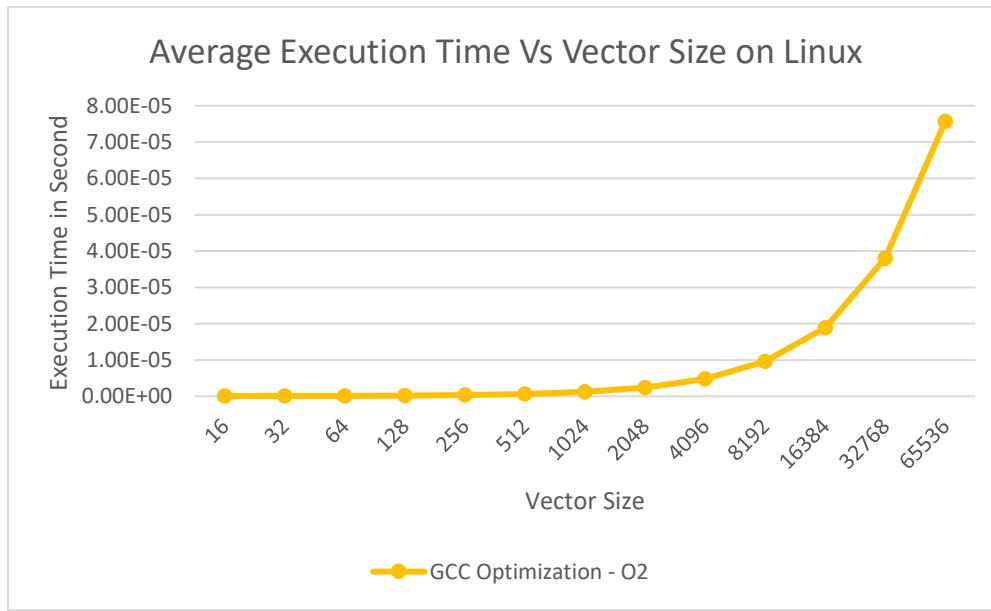
Figure 78 - run the C++ code with O2 Optimization

Graph

On the Table and graph below, we can see the execution time in Linux for O2 optimization after we run the code.

Table 11 - Execution time for Compiler Generated Optimization with O2

| Vector Size | Execution time in second for GCC Optimization with O2 |
|-------------|---|
| 16 | 5.44E-08 |
| 32 | 6.27E-08 |
| 64 | 9.91E-08 |
| 128 | 1.74E-07 |
| 256 | 3.21E-07 |
| 512 | 6.15E-07 |
| 1024 | 1.20E-06 |
| 2048 | 2.37E-06 |
| 4096 | 4.72E-06 |
| 8192 | 9.57E-06 |
| 16384 | 1.88E-05 |
| 32768 | 3.79E-05 |
| 65536 | 7.57E-05 |



Graph 9 - Execution time for Compiler Generated Optimization with O2

Complete Assembly Code for O2

Below we can see the O2 optimization complete assembly Code in Linux.

```

1 .file  "Peiravani_Paniz_dotProduct.cpp"
2 .text
3 .globl _Z10dotProductRKSt6vectorIfSaIfEEES3_i
4 .type _Z10dotProductRKSt6vectorIfSaIfEEES3_i, @function
5 _Z10dotProductRKSt6vectorIfSaIfEEES3_i:
6 .LFB865:
7     .cfi_startproc
8     endbr64
9     pushq %rbp
10    .cfi_offset 6, -16
11    movq %rsp, %rbp
12    .cfi_offset 6, -16
13    subq $48, %rsp
14    movq %rdi, -24(%rbp)
15    movq %rsi, -32(%rbp)
16    movl %edx, -36(%rbp)
17    pxor %xmm0, %xmm0
18    movss %xmm0, -8(%rbp)
19    movl $0, -4(%rbp)
20    jmp .L2
21 .L3:
22     movl -4(%rbp), %eax
23     movslq %eax, %rdx
24     movq -24(%rbp), %rax
25     movq %rdx, %rsi
26     movq %rax, %rdi
27     call _ZNKSt6vectorIfSaIfEEixEm
28     movss (%rax), %xmm2
29     movss %xmm2, -40(%rbp)
30     movl -4(%rbp), %eax
31     movslq %eax, %rdx
32     movq -32(%rbp), %rax
33

```

Figure 79 - O2 optimization on Linux

```

33     movq -32(%rbp), %rax
34     movq %rdx, %rsi
35     movq %rax, %rdi
36     call _ZNKSt6vectorIfSaIfEEixEm
37     movss (%rax), %xmm0
38     mulss -40(%rbp), %xmm0
39     movss -8(%rbp), %xmm1
40     addss %xmm1, %xmm0
41     movss %xmm0, -8(%rbp)
42     addl $1, -4(%rbp)
43 .L2:
44     movl -4(%rbp), %eax
45     cmpl -36(%rbp), %eax
46     jl .L3
47     movss -8(%rbp), %xmm0
48     leave
49     .cfi_def_cfa 7, 8
50     ret
51     .cfi_endproc
52 .LFE865:
53     .size _Z10dotProductRKSt6vectorIfSaIfEEES3_i, .-_Z10dotProductRKSt6vectorIfSaIfEEES3_i
54     .section .text._ZNKSt6vectorIfSaIfEEixEm,"axG",@progbits,_ZNKSt6vectorIfSaIfEEixEm,comdat
55     .align 2
56     .weak _ZNKSt6vectorIfSaIfEEixEm
57     .type _ZNKSt6vectorIfSaIfEEixEm, @function
58 _ZNKSt6vectorIfSaIfEEixEm:
59 .LFB868:
60     .cfi_startproc
61     endbr64
62     pushq %rbp
63     .cfi_offset 6, -16
64     .cfi_offset 6, -16
65     movq %rbp, %rbp

```

Figure 80 - O2 optimization on Linux

```

55      movq    %rsp, %rbp
56      .cfi_def_cfa_register 6
57      movq    %rdi, -8(%rbp)
58      movq    %rsi, -16(%rbp)
59      movq    -8(%rbp), %rax
60      movq    (%rax), %rdx
61      movq    -16(%rbp), %rax
62      salq    $2, %rax
63      addq    %rdx, %rax
64      popq    %rbp
65      .cfi_def_cfa 7, 8
66      ret
67      .cfi_endproc
68 .LFE868:
69     .size   _ZNKSt6vectorIfSaIfEEixEm, .._ZNKSt6vectorIfSaIfEEixEm
70     .ident  "GCC: (Ubuntu 11.3.0-1ubuntu1-22.04) 11.3.0"
71     .section .note.GNU-stack,"",@progbits
72     .section .note.gnu.property,"a"
73     .align 8
74     .long   1f - 0f
75     .long   4f - 1f
76     .long   5
77 0:
78     .string "GNU"
79 1:
80     .align 8
81     .long   0xc0000002
82     .long   3f - 2f
83 2:
84     .long   0x3
85 3:
86     .align 8
87 4:

```

Figure 81 - O2 optimization on Linux

Manually Optimization

Now we are manually optimizing the code. The red lines show the code that I can remove and the green lines the code that I optimized. One way to optimize is to reduce the use of `mov` instructions by reusing registers or rearranging instructions and changing bytes storage for memory usage and speed.

```

.file  "Peiravani_Paniz_dotProduct.cpp"
.text
.p2align
.globl _Z10dotProductRKSt6vectorIfSaIfEES3_i
.type  _Z10dotProductRKSt6vectorIfSaIfEES3_i, @function
_Z10dotProductRKSt6vectorIfSaIfEES3_i:
.LFB865:
    .cfi_startproc
    endbr64
    testl %edx, %edx
    jle  .L10
    jle  .L4
    leal  -1(%rdx), %eax
    movq  (%rdi), %rd
    movq  (%rsi), %rcx
    cmpl  $6, %eax
    jbe  .L11
    movl  %edx, %esi
    movslq %edx, %rdx
    xorl  %eax, %eax
    vxorps %xmm0, %xmm0, %xmm0
    vxorps %xmm1, %xmm1, %xmm1
    shr1  $3, %esi
    salq  $5, %rsi
    .p2align 4,,10
    .p2align 3

```

```

.L4:
    vmovups    (%rdi,%rax), %ymm4
    vmulps (%rcx,%rax), %ymm4, %ymm1
    addq   $32, %rax
    vaddss %xmm1, %xmm0, %xmm0
    vshufps   $85, %xmm1, %xmm1, %xmm3
    vshufps   $255, %xmm1, %xmm1, %xmm2
    vaddss %xmm3, %xmm0, %xmm0
    vunpckhps %xmm1, %xmm1, %xmm3
    vextractf128 $0x1, %ymm1, %xmm1
    vaddss %xmm3, %xmm0, %xmm0
    vaddss %xmm2, %xmm0, %xmm0
    vshufps   $85, %xmm1, %xmm1, %xmm2
    vaddss %xmm1, %xmm0, %xmm0
    vaddss %xmm2, %xmm0, %xmm0
    vunpckhps %xmm1, %xmm1, %xmm2
    vshufps   $255, %xmm1, %xmm1, %xmm1
    vaddss %xmm2, %xmm0, %xmm0
    vaddss %xmm1, %xmm0, %xmm0
    cmpq   %rsi, %rax
    jne    .L4
    movl   %edx, %esi
    andl   $-8, %esi
    movl   %esi, %eax
    cmpl   %esi, %edx
    je     .L16
    vzeroupper

.L3:
    vmovss  (%rdi,%rax,4), %xmm0
    movl   %edx, %r8d
    subl   %esi, %r8d
    leal   -1(%r8), %r9d
    cmpl   $2, %r9d
    jbe    .L7
    vmovups    (%rdi,%rsi,4), %xmm5
    vmulps (%rcx,%rsi,4), %xmm5, %xmm1
    movl   %r8d, %esi
    andl   $-4, %esi
    addl   %esi, %eax
    vaddss %xmm1, %xmm0, %xmm0
    vshufps   $85, %xmm1, %xmm1, %xmm2
    vaddss %xmm2, %xmm0, %xmm0
    vunpckhps %xmm1, %xmm1, %xmm2
    vshufps   $255, %xmm1, %xmm1, %xmm1
    vaddss %xmm2, %xmm0, %xmm0
    vaddss %xmm1, %xmm0, %xmm0
    cmpl   %esi, %r8d
    je     .L1

.L7:
    cltq
    .p2align 4,,10
    .p2align 3

.L9:
    vmovss  (%rdi,%rax,4), %xmm1
    vmulss (%rcx,%rax,4), %xmm1, %xmm1

```

```

vmulss (%rcx,%rax,4), %xmm0, %xmm0
addq $1, %rax
vaddss %xmm1, %xmm0, %xmm0
vaddss %xmm0, %xmm1, %xmm1
cmpl %eax, %edx
cmpq %rax, %rdx
jg .L9
jg .L3
vmovaps %xmm1, %xmm0
ret
.p2align 4,,10
.p2align 3
.L10:
vxorps %xmm0, %xmm0, %xmm0
.L1:
ret
.p2align 4,,10
.p2align 3
.L16:
vzeroupper
ret
.L11:
xorl %esi, %esi
xorl %eax, %eax
vxorps %xmm0, %xmm0, %xmm0
jmp .L3
.cfi_endproc
.LFE865:
.size _Z10dotProductRKSt6vectorIfSaIfEES3_i, .-
_Z10dotProductRKSt6vectorIfSaIfEES3_i
.ident "GCC: (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0"
.section .note.GNU-stack,"",@progbits
.section .note.gnu.property,"a"
.align 8
.long 1f - 0f
.long 4f - 11
.long 5
0:
.string "GNU"
1:
.align 8
.long 0xc0000002
.long 3f - 2f
2:
.long 0x3
3:
.align 8
.L4:
vxorps %xmm1, %xmm1, %xmm1
vmovaps %xmm1, %xmm
ret
.cfi_endproc

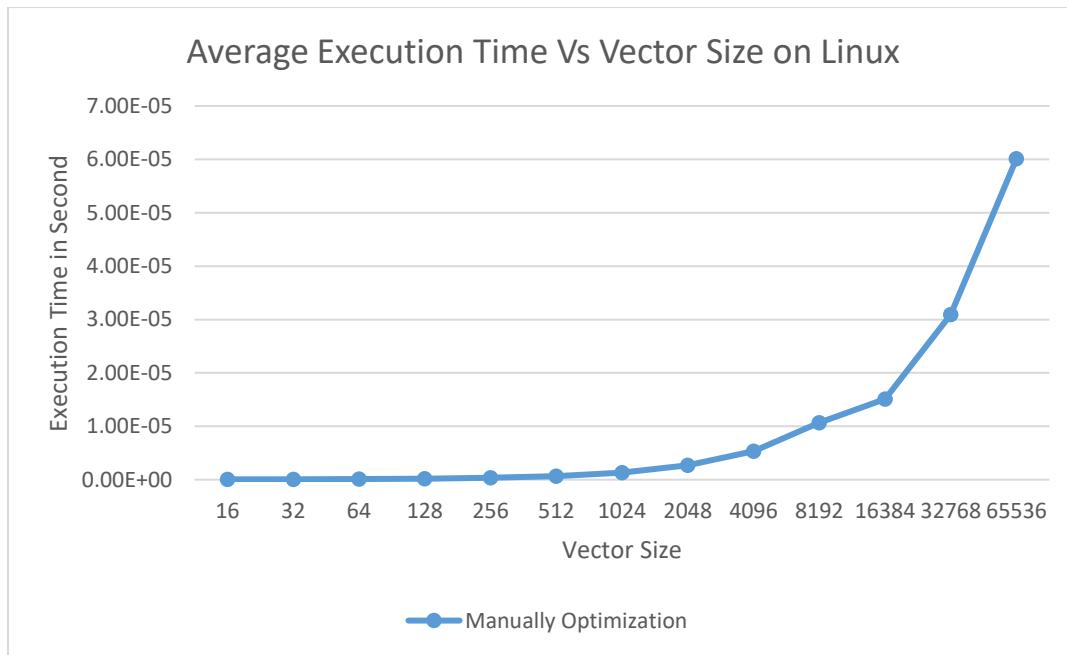
```

Graph

And after we run the code, we can see that the execution time is less than before. Below table shows the execution time of our manually optimization code.

Table 12 - Execution Time for Manually Optimization on Linux

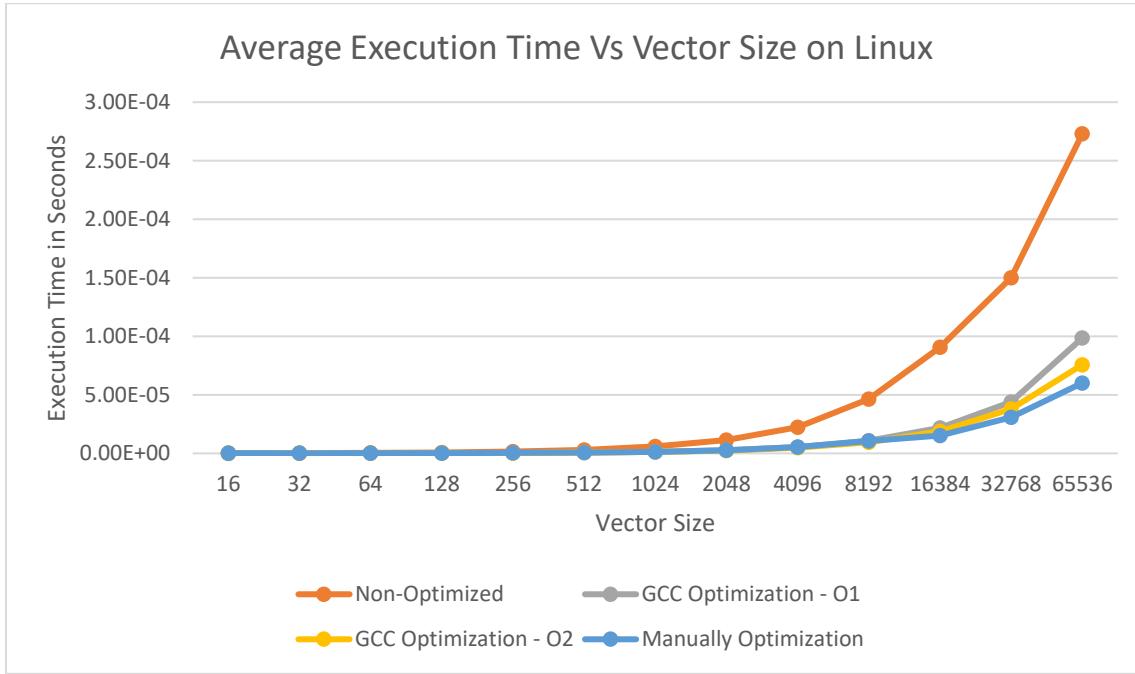
| Vector Size | Execution time in second for Manually Optimized on Linux |
|-------------|--|
| 16 | 7.06E-08 |
| 32 | 7.04E-08 |
| 64 | 1.08E-07 |
| 128 | 1.86E-07 |
| 256 | 3.47E-07 |
| 512 | 6.84E-07 |
| 1024 | 1.35E-06 |
| 2048 | 2.70E-06 |
| 4096 | 5.35E-06 |
| 8192 | 1.07E-05 |
| 16384 | 1.51E-05 |
| 32768 | 3.09E-05 |
| 65536 | 6.01E-05 |



Graph 10 - Execution Time for Manually Optimization on Linux

Graph

Now that we have our execution time for non-optimized, O1 compiler optimization, O2 compiler optimization, and manually optimization we can put the execution time on graph to compare the executions times on Linux, Ubuntu.

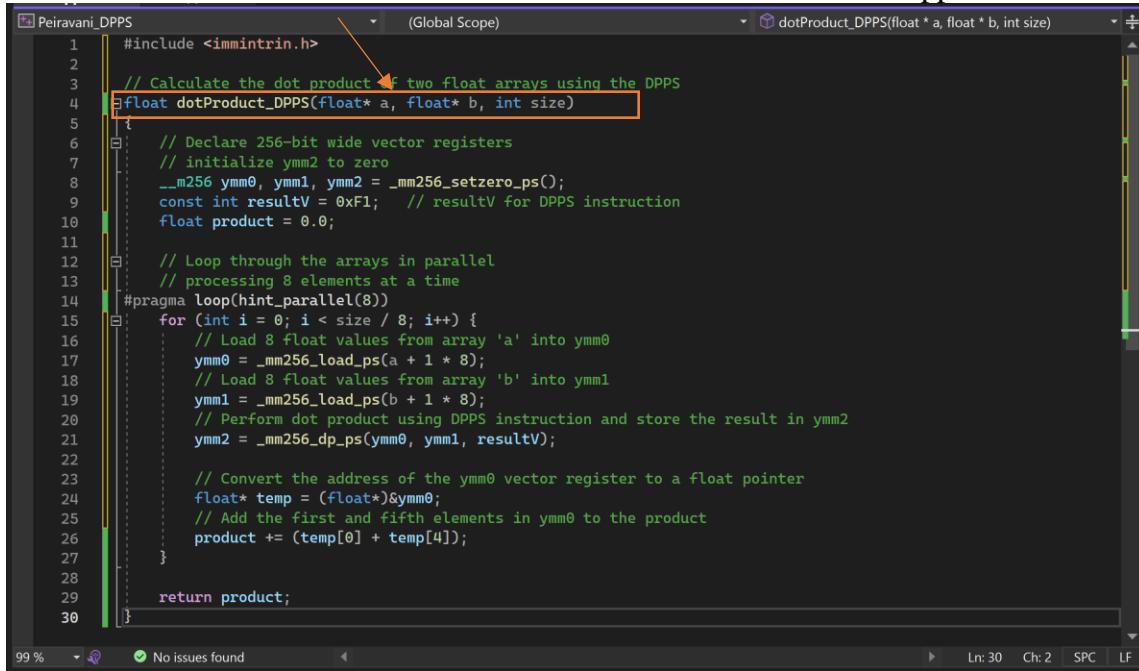


Graph 11 - Task 3 for Linux

Task 4 – DPPS

Windows – Intel x64

Now we are going to use Determinantal point processes, DPPS. DPPS vector instruction will optimize the code further. In other words, DPPS has better performance, and it is faster. We will use DPPS in our *dotProduct.cpp* since in that file out dot Products computers. Below is our DPPS code for our dotProcut. The only update that we need to out main.cpp is that to change the format that it calls dotProduct to be same as the one in out dotProduct.cpp.



```

1 #include <immintrin.h>
2
3 // Calculate the dot product of two float arrays using the DPPS
4 float dotProduct_DPPS(float* a, float* b, int size)
5 {
6     // Declare 256-bit wide vector registers
7     // initialize ymm2 to zero
8     __m256 ymm0, ymm1, ymm2 = _mm256_setzero_ps();
9     const int resultV = 0xF1; // resultV for DPPS instruction
10    float product = 0.0;
11
12    // Loop through the arrays in parallel
13    // processing 8 elements at a time
14    #pragma loop(hint_parallel(8))
15    for (int i = 0; i < size / 8; i++) {
16        // Load 8 float values from array 'a' into ymm0
17        ymm0 = _mm256_load_ps(a + i * 8);
18        // Load 8 float values from array 'b' into ymm1
19        ymm1 = _mm256_load_ps(b + i * 8);
20        // Perform dot product using DPPS instruction and store the result in ymm2
21        ymm2 = _mm256_dp_ps(ymm0, ymm1, resultV);
22
23        // Convert the address of the ymm0 vector register to a float pointer
24        float* temp = (float*)&ymm0;
25        // Add the first and fifth elements in ymm0 to the product
26        product += (temp[0] + temp[4]);
27    }
28
29    return product;
30}

```

Graph 12 - DPPS

Run the Code

After we run the code, we can see that the execution time is less than previous steps.

```

Vector size: 16
Average time: 4e-08
-----
Vector size: 32
Average time: 8e-08
-----
Vector size: 64
Average time: 1.2e-07
-----
Vector size: 128
Average time: 1.3e-07
-----
Vector size: 256
Average time: 1.9e-07
-----
Vector size: 512
Average time: 2.3e-07
-----
Vector size: 1024
Average time: 2.6e-07
-----
Vector size: 2048
Average time: 2.8e-07
-----
Vector size: 4096
Average time: 3.4e-07
-----
Vector size: 8192
Average time: 4.1e-07
-----
Vector size: 16384
Average time: 4.6e-07
-----
Vector size: 32768
Average time: 5e-07
-----
```

Figure 82 - DPPS Result

```
-----
Vector size: 32768
Average time: 5e-07
-----
Vector size: 65536
Average time: 5.4e-07
```

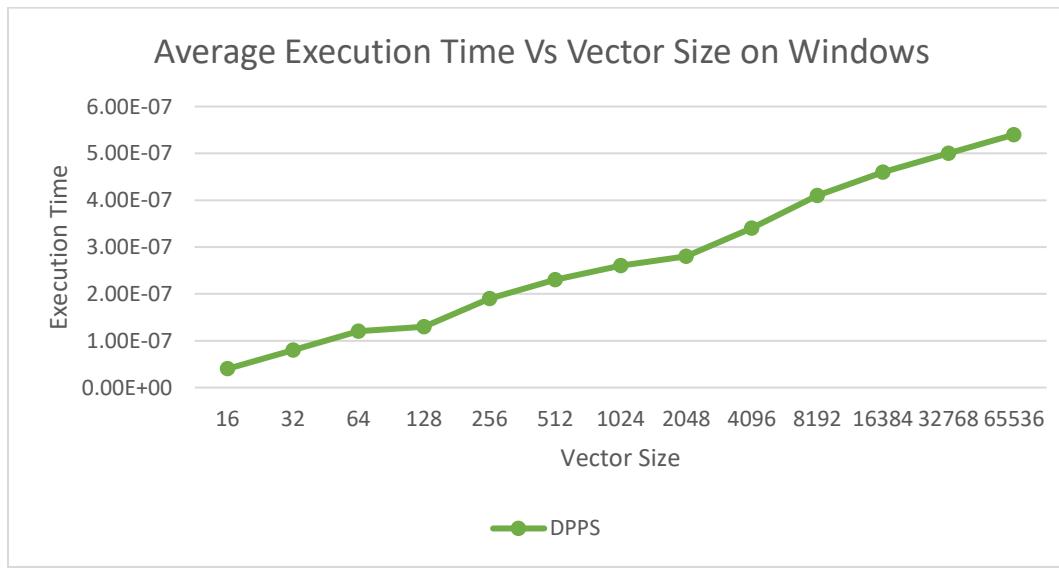
Figure 83 - DPPS Result

Graph

Now that we have execution time, we will graph Average Execution time Vs vector size for unoptimized code. We ran the code 10 times for each vector size to be able to find the average execution time that is more accurate.

Table 13 - DPPS for Windows

| Vector Size | Execution time in seconds for DPPS on Windows |
|-------------|---|
| 16 | 4.00E-08 |
| 32 | 8.00E-08 |
| 64 | 1.20E-07 |
| 128 | 1.30E-07 |
| 256 | 1.90E-07 |
| 512 | 2.30E-07 |
| 1024 | 2.60E-07 |
| 2048 | 2.80E-07 |
| 4096 | 3.40E-07 |
| 8192 | 4.10E-07 |
| 16384 | 4.60E-07 |
| 32768 | 5.00E-07 |
| 65536 | 5.40E-07 |



Graph 13 - DPPS for Windows

Linux – Ubuntu

Now we are going to use Determinantal point processes, DPPS. DPPS vector instruction will optimize the code further. In other words, DPPS has better performance, and it is faster. We will use DPPS in our *dotProduct.cpp* since in that file out dot Products computers. Below is our DPPS code for our dotProcut. The only update that we need to out main.cpp is that change the format that it calls dotProduct to be same as the one in out dotProduct.cpp.

```

1 #include <immintrin.h>
2 #include <stdio.h>
3
4 // Calculate the dot product of two float arrays using the DPPS instruction
5 float dotProduct_DPPS(float* a, float* b, int size)
6 {
7     // Declare 256-bit wide vector registers and initialize ymm2 to zero
8     _mm256 ymm0, ymm1, ymm2 = _mm256_setzero_ps();
9     const int resultV = 0xF1; // resultV for DPPS instruction
10    float product = 0.0;
11
12 // Loop through the arrays in parallel, processing 8 elements at a time
13 #pragma loop(hint_parallel(8))
14    for (int i = 0; i < size / 8; i++) {
15        // Load 8 float values from array 'a' into ymm0
16        ymm0 = _mm256_loadu_ps(a + i * 8);
17        // Load 8 float values from array 'b' into ymm1
18        ymm1 = _mm256_loadu_ps(b + i * 8);
19        // Perform dot product using DPPS instruction and store the result in ymm2
20        ymm2 = _mm256_dp_ps(ymm0, ymm1, resultV);
21
22        // Convert the address of the ymm2 vector register to a float pointer
23        float* temp = (float*)&ymm2;
24        // Add the first and fifth elements in ymm2 to the product
25        product += (temp[0] + temp[4]);
26    }
27
28    return product;
29 }
```

Figure 84 – DPPS

Run the Code

In order for our code to run, we need to use AVX instruction which we know our CPU supports based on task 1.

```

paniz@paniz-VirtualBox:~/Desktop/DPPS$ g++ -favx main.cpp dpps.cpp
-o DPPS
paniz@paniz-VirtualBox:~/Desktop/DPPS$ ./DPPS
```

Figure 85 - Run DPPS on Linux

```

Vector size: 16
Average time: 5.87e-08
-----
Vector size: 32
Average time: 5.71e-08
-----
Vector size: 64
Average time: 7.02e-08
-----
Vector size: 128
Average time: 1.008e-07
-----
Vector size: 256
Average time: 1.057e-07
-----
Vector size: 512
Average time: 3.086e-07
-----
Vector size: 1024
Average time: 5.668e-07
-----
Vector size: 2048
Average time: 1.1153e-06
-----
Vector size: 4096
Average time: 2.1556e-06
-----
```

Figure 86 - DPPS result

```

Vector size: 8192
Average time: 4.4401e-06
-----
Vector size: 16384
Average time: 8.8926e-06
-----
Vector size: 32768
Average time: 2.58451e-05
-----
Vector size: 65536
Average time: 4.08041e-05

```

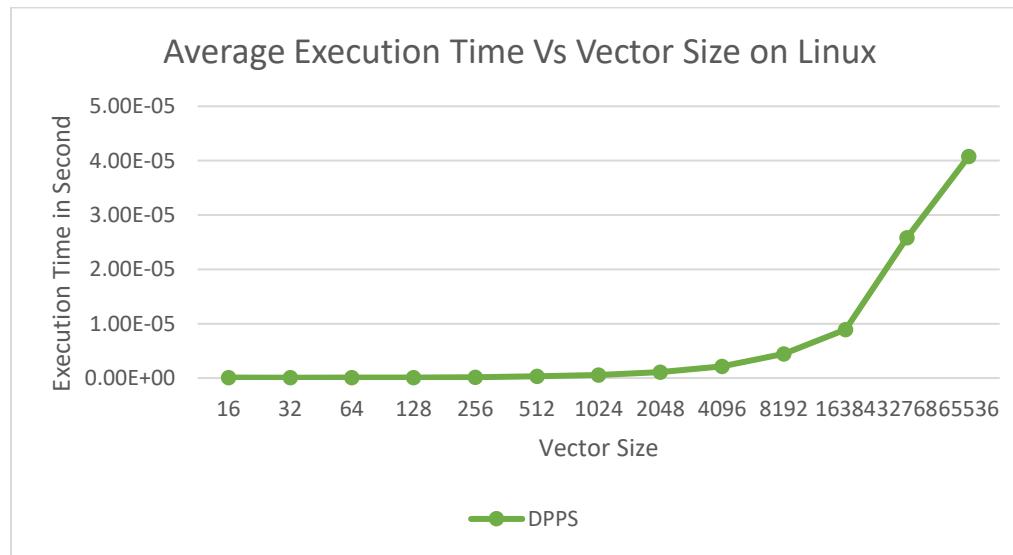
Figure 87 - DPPs result

Graph

Now that we have execution time, we will graph Average Execution time Vs vector size for unoptimized code. We ran the code 10 times for each vector size to be able to find the average execution time that is more accurate.

Table 14 - DPPs on Linux

| Vector Size | Execution time in seconds for DPPs on Linux |
|-------------|---|
| 16 | 5.87E-08 |
| 32 | 5.71E-08 |
| 64 | 7.02E-08 |
| 128 | 1.01E-07 |
| 256 | 1.66E-07 |
| 512 | 3.09E-07 |
| 1024 | 5.67E-07 |
| 2048 | 1.12E-06 |
| 4096 | 2.16E-06 |
| 8192 | 4.44E-06 |
| 16384 | 8.89E-06 |
| 32768 | 2.58E-05 |
| 65536 | 4.08E-05 |

*Graph 14 - DPPs Graph*

Task 5 – All Plots in on Graph

Windows – Intel x64

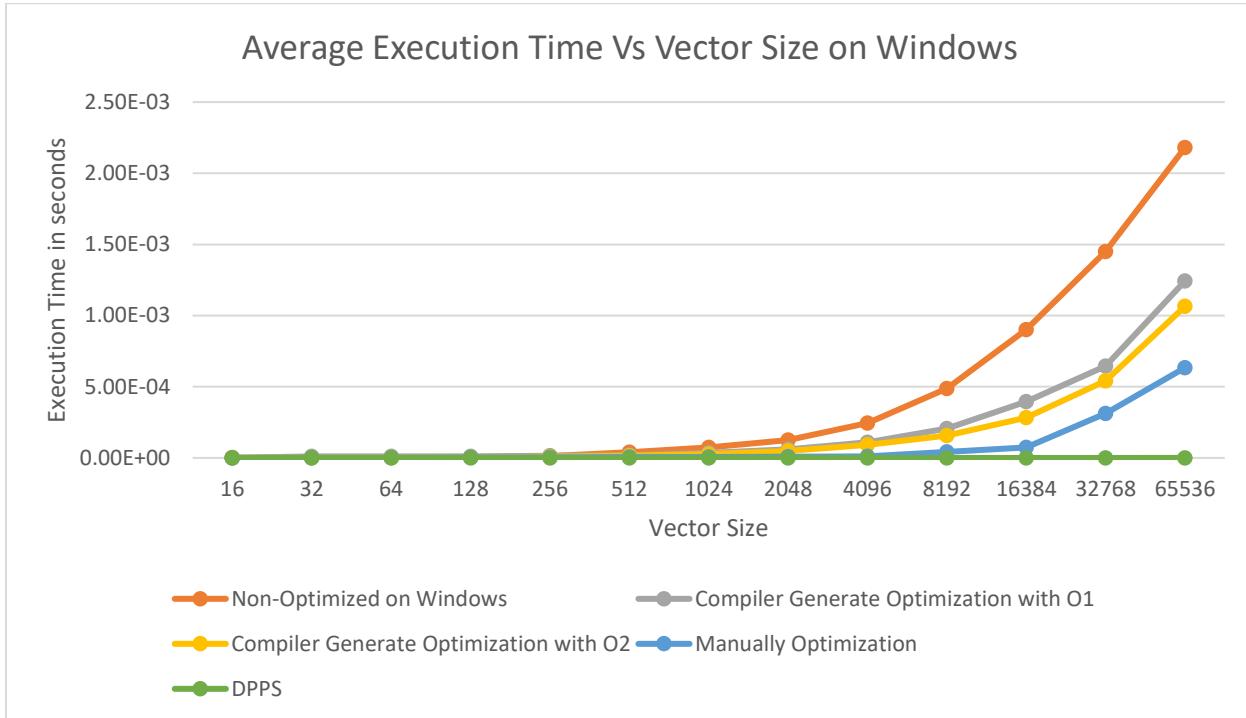
Now that we have finished tasks 1 to 4, we are going to compare all of the result in one graph. Below you can see the table that includes all the execution times that we found out for each task in windows.

Table 15 – Windows Intel x64

| Vector Size | Non-Optimized | Compiler Optimization - O1 | Compiler Optimization - O2 | Manually Optimization | DPPS |
|-------------|---------------|----------------------------|----------------------------|-----------------------|----------|
| 16 | 8.20E-07 | 6.90E-07 | 6.60E-07 | 4.70E-07 | 4.00E-08 |
| 32 | 2.04E-06 | 9.97E-06 | 1.55E-06 | 1.97E-06 | 8.00E-08 |
| 64 | 3.84E-06 | 1.08E-05 | 2.88E-06 | 2.46E-06 | 1.20E-07 |
| 128 | 6.71E-06 | 1.17E-05 | 4.64E-06 | 4.19E-06 | 1.30E-07 |
| 256 | 1.28E-05 | 1.50E-05 | 8.25E-06 | 4.90E-06 | 1.90E-07 |
| 512 | 4.05E-05 | 2.00E-05 | 1.44E-05 | 6.05E-06 | 2.30E-07 |
| 1024 | 7.33E-05 | 3.64E-05 | 2.68E-05 | 7.15E-06 | 2.60E-07 |
| 2048 | 1.25E-04 | 6.06E-05 | 4.82E-05 | 9.10E-06 | 2.80E-07 |
| 4096 | 2.44E-04 | 1.10E-04 | 9.27E-05 | 1.16E-05 | 3.40E-07 |
| 8192 | 4.87E-04 | 2.07E-04 | 1.57E-04 | 4.23E-05 | 4.10E-07 |
| 16384 | 9.00E-04 | 3.95E-04 | 2.82E-04 | 7.45E-05 | 4.60E-07 |
| 32768 | 1.45E-03 | 6.46E-04 | 5.42E-04 | 3.12E-04 | 5.00E-07 |
| 65536 | 2.18E-03 | 1.24E-03 | 1.06E-03 | 6.34E-04 | 5.40E-07 |

If you look at the graph, we can see the the order of execution time from fast to slow is from Non-Optimization, Compiler optimization - O1, Compiler Optimization - O2, manually Optimization, and DPPS

Table 16 – Windows Intel x64



Linux – Ubuntu

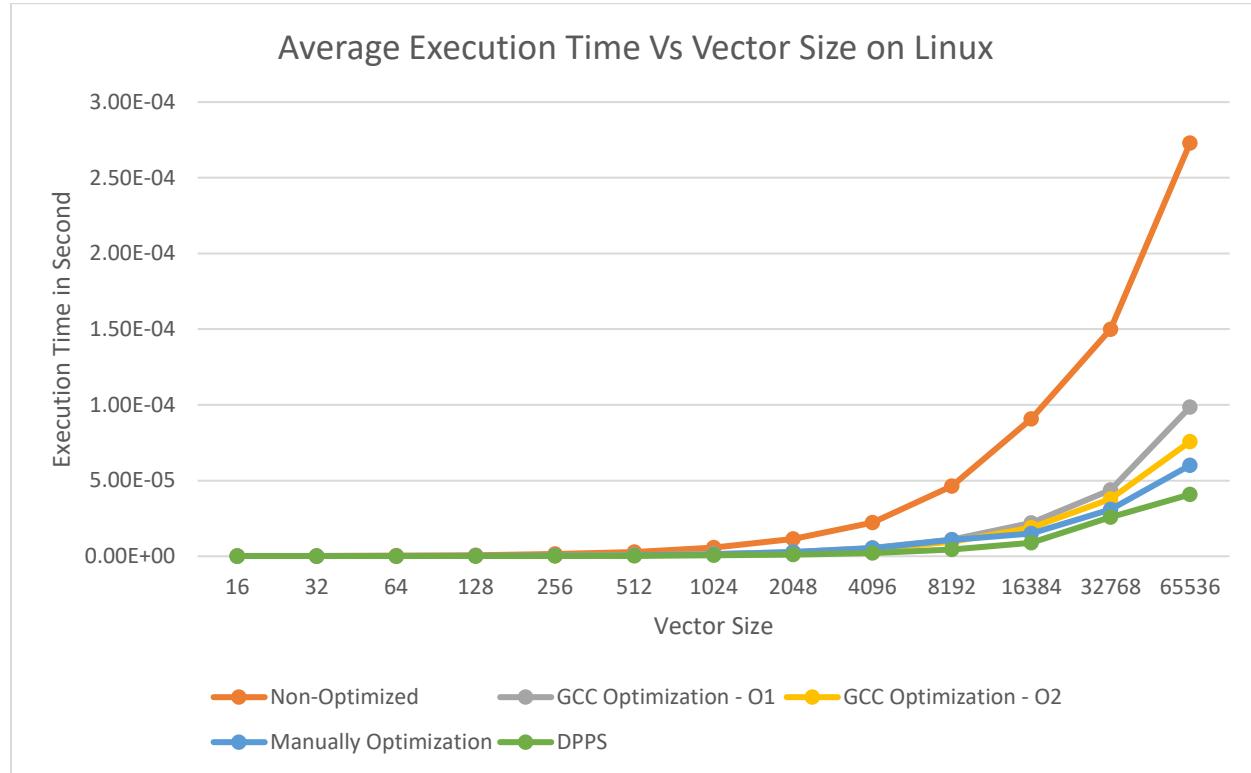
Now that we have finished tasks 1 to 4, we are going to compare all of the result in one graph. Below you can see the table that includes all the execution times that we found out for each task in Linux.

Table 17 – Linux, Ubuntu

| Vector Size | Non-Optimized | GCC Optimization - O1 | GCC Optimization - O2 | Manually Optimization | DPPS |
|-------------|---------------|-----------------------|-----------------------|-----------------------|----------|
| 16 | 2.28E-07 | 5.26E-08 | 5.44E-08 | 7.06E-08 | 5.87E-08 |
| 32 | 2.17E-07 | 6.44E-08 | 6.27E-08 | 7.04E-08 | 5.71E-08 |
| 64 | 3.97E-07 | 1.08E-07 | 9.91E-08 | 1.08E-07 | 7.02E-08 |
| 128 | 7.36E-07 | 1.93E-07 | 1.74E-07 | 1.86E-07 | 1.01E-07 |
| 256 | 1.43E-06 | 3.66E-07 | 3.21E-07 | 3.47E-07 | 1.66E-07 |
| 512 | 2.82E-06 | 7.23E-07 | 6.15E-07 | 6.84E-07 | 3.09E-07 |
| 1024 | 5.83E-06 | 1.39E-06 | 1.20E-06 | 1.35E-06 | 5.67E-07 |
| 2048 | 1.14E-05 | 2.76E-06 | 2.37E-06 | 2.70E-06 | 1.12E-06 |
| 4096 | 2.23E-05 | 5.50E-06 | 4.72E-06 | 5.35E-06 | 2.16E-06 |
| 8192 | 4.63E-05 | 1.10E-05 | 9.57E-06 | 1.07E-05 | 4.44E-06 |
| 16384 | 9.07E-05 | 2.19E-05 | 1.88E-05 | 1.51E-05 | 8.89E-06 |
| 32768 | 1.50E-04 | 4.38E-05 | 3.79E-05 | 3.09E-05 | 2.58E-05 |
| 65536 | 2.73E-04 | 9.85E-05 | 7.57E-05 | 6.01E-05 | 4.08E-05 |

If you look at the graph, we can see the the order of execution time from fast to slow is from Non-Optimization, Compiler optimization - O1, Compiler Optimization - O2, manually Optimization, and DPPS

Table 18 - Linux, Ubuntu



Conclusion

In conclusion, this final take-home assignment focused on optimizing compiler-generated code for computing dot products using vector instructions. This take-home test needed to be implemented on two different operating system. I used Windows intel x64 and Linux, Ubuntu. First, I determine my CPU ID to see which vector instructions I can use that my CPU will support. Next, I generated a C++ code which has two separate files. One for main function and one for calculating the dot product. First, I disable automatic parallelization and automatic vectorization to observe the output. After that I enable them to see the output for optimizations. For optimization, I used O1, O2 flags and then I manually optimized the assembly code myself as well. I also used DPPs for more optimizations. For each task, I found the execution time. I ran the code times and saved the average execution times so the result be more accurate. Lastly, I compare all the graphs for each task to observe how does optimization works. Overall, From the graph, I observed that the order of execution time from fastest to slowest was Non-Optimization, Compiler optimization - O1, Compiler Optimization - O2, manually Optimization, and DPPS.